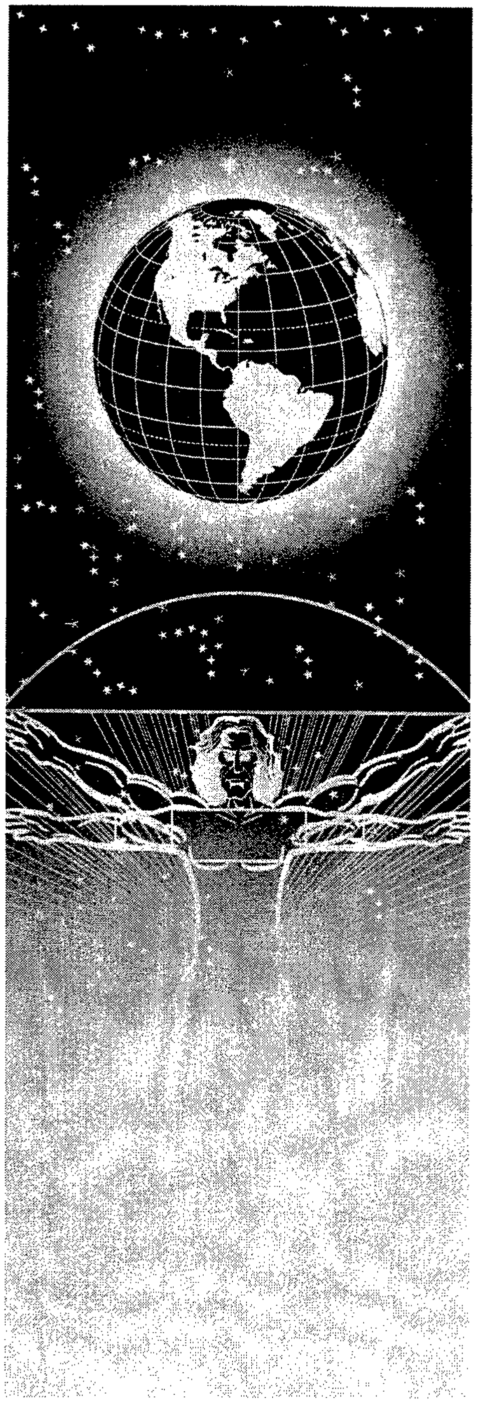
AFRL-HE-WP-TR-2002-0184



UNITED STATES AIR FORCE RESEARCH LABORATORY

TECHNIQUES FOR PREPROCESSING SPEECH SIGNALS FOR MORE EFFECTIVE AUDIO INTERFACES

PHILLIP L. DELEON

NEW MEXICO STATE UNIVERSITY
KLIPSCH SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
BOX 30001/DEPT. 3-0
LAS CRUCES NM 88003-8001

DECEMBER 2001

FINAL REPORT FOR THE PERIOD JANUARY 2000 TO DECEMBER 2001

Human Effectiveness Directorate
Crew System Interface Division
2255 H Street
Wright-Patterson AFB OH 45433-7022

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE December 2001	3. REPORT TYPE AND DATES COVERED Final Report, January 2000 - December 2001

4. TITLE AND SUBTITLE Techniques for Preprocessing Speech Signals for More Effective Audio Interfaces	5. FUNDING NUMBERS C:F41624-99-1-0001 PE: 62202F PR: 7184 TA: 10 WU: NM
6. AUTHOR(S) Phillip L. DeLeon	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) New Mexico State University Klipsch School of Electrical and Computer Engineering Box 30001/ Dept. 3-0 Las Cruces NM 88003-8001	8. PERFORMING ORGANIZATION
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory Human Effectiveness Directorate Crew System Interface Division Air Force Materiel Command 2255 H Street, Bldg 248 Wright-Patterson AFB OH 45433-7022	10. SPONSORING/MONITORING AFRL-HE-WP-TR-2002-0184
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A user receiving instructions from various speech sources may be overwhelmed when these sources are speaking simultaneously in an unfavorable acoustical and noisy environment. In such a case, the user is required to separate the various sources from the mixture in order to make the speech intelligible. If no one source dominates or the mixing occurs for a sustained period of time, the human user may become mentally and physically overloaded resulting in fatigue and thus failing to separate the various speech sources into intelligible signals. In the case of the speech recognizer, recognition accuracy may be degraded to unacceptable levels. In this final report in research into methods for blind enhancement and separation of mixtures of speech signals, we present our results form further development and refinement of Frequency Domain, Second-Order Statistics-based decorrelation algorithms and in particular the Multi-resolution Frequency-Domain algorithm. These results include modifications to the algorithm for better performance at reduced cost, implementation, and performance evaluation under a wide set of noise and acoustic environments. Finally, we report on the development of a publically-available database specifically designed for evaluating various speech enhancement/separation algorithms.

14. SUBJECT TERMS Blind Speech Separation Audio Interfaces	15. NUMBER OF PAGES 97
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UNLIMITED

This page intentionally left blank

Contents

List of Figures

List of Tables

1 Summary

A user (human or computer-based speech recognizer) receiving instructions from various speech sources (human or synthesized) may be overwhelmed when these sources are speaking simultaneously in an unfavorable acoustical and noisy environment. In such a case, the user is required to separate the various sources from the mixture in order to make the speech intelligible. If no one source dominates or the mixing occurs for a sustained period of time, the human user may become mentally and physically overloaded resulting in fatigue and thus failing to separate the various speech sources into intelligible signals. In the case of the speech recognizer, recognition accuracy may be degraded to unacceptable levels.

In this research, we have gained an understanding of the source separation problem for convolutional mixtures and developed expertise with the Frequency-Domain, Second-Order Statistics (FDSOS) based blind speech separation (BSS) algorithms. As a starting point, we have implemented a recently-developed version called the Multiresolution Frequency-Domain (MFRD) algorithm and analyzed its performance. We have developed several modifications to the MRFD algorithm for better performance at reduced cost, implemented it, and evaluated it under a wide set of noise and acoustic environments in both simulated and real-world tests. Modifications include methods for exploiting frequency-domain symmetries in order to reduce computation by 50% and initialization strategies for improved performance. We have experimentally determined optimal parameters for the algorithm and studied the effects of room size and liveliness on separation performance. Simulations with both digitally-filtered and mixed speech signals as well as real-world signals indicate good performance with this algorithm with 10-20dB improvement to the Signal-to-Interference Ratio (SIR) of these sources.

In the important case of speech enhancement, i.e. blind suppression of background noise from a desired speech signal, we have been able to increase Signal-to-Interference Ratio Improvement (SIRI) upwards of 20dB. This may have significant impact on the problem of automatic speech recognition in noisy environments. Our work has also demonstrated a robustness to separating speech signals in the presence of noise. The potential of these algorithms for real-time processing has been examined and we have concluded that good performance is possible but with at least a 2sec. delay due to the inherent "block processing" nature of the algorithm. Finally, we have developed a blind speech signal corpus or database containing speech signal mixtures in a variety of acoustical environments. This database is now available on the Internet at

`http://www.ece.nmsu.edu/~pdeleon/BSS/index.html`

for free download.

2 Introduction

In many audio-interface, multimedia, and speech recognition applications, mixtures of speech signals from various competing acoustic sources (other speakers and noise) must be separated out before processing [1]. Given the complicated nature of speech signals (non-stationary, overlap in time- and frequency-domains, etc...) this is a difficult problem compounded by environmental effects such as noise and reverberation and a strong desire for a simple algorithm suitable for real-time operation. Several methods have been proposed some of which have shown moderate success at various aspects of the problem but often at the expense of high computational complexity [2],[3].

The basic problem (two channel case) is illustrated in Fig. 1 and described as follows. Let s_1, s_2 be two unknown source speech signals and x_1, x_2 be two mixture signals each composed of the sum of the acoustically-filtered (echoed and reverberated) source signals; the impulse response from source i to microphone j is given by h_{ji}. The problem is this: given *only* the received signals (mixtures of filtered speech), x_1, x_2, produce two signals y_1, y_2 which approximate the individual source signals. Note we have no information regarding the original sources or acoustical filters. The "blind" designation denotes the fact we are given no *a priori* information to use in the separation task.

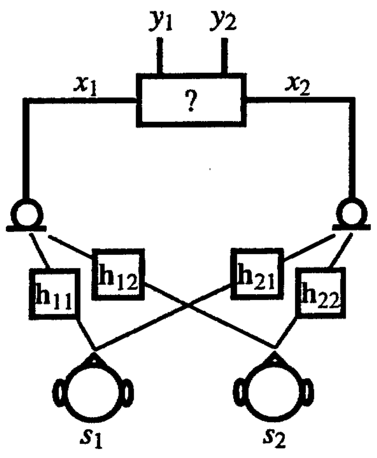


Figure 1: Block diagram of speech separation problem.

The mixture signals in Fig. 1 can be expressed as a convolution, i.e.,

$$x_j(n) = \sum_{i=1}^{2}\sum_{p=0}^{P-1} h_{ji}(p)s_i(n-p), j = 1, 2 \tag{2.1}$$

where $h_{ji}(p)$ models the P-point impulse response from source i to microphone j. The two mixture signals in (2.1) can be stacked in vector form and written as

$$\begin{bmatrix} x_1(n) \\ x_2(n) \end{bmatrix} = \begin{bmatrix} h_{11}(n) & h_{12}(n) \\ h_{21}(n) & h_{22}(n) \end{bmatrix} * \begin{bmatrix} s_1(n) \\ s_2(n) \end{bmatrix} \tag{2.2}$$

or simply as,

$$\mathbf{x}(n) = \mathbf{H}(n) * \mathbf{s}(n), \tag{2.3}$$

where $\mathbf{x}(n) = [x_1(n)\ x_2(n)]^T$ is the vector of received mixture signals, $\mathbf{H}(n)$ is the 2×2 mixing filter matrix whose elements are h_{ji}, $*$ is the convolution operator, and $\mathbf{s}(n) = [s_1(n)\ s_2(n)]^T$ is the source signal vector. The aim of the BSS method is to find a 2×2 un-mixing filter matrix, $\mathbf{W}(n)$, where each element is of length Q, that separates the two sources up to an arbitrary filter and permutation, i.e.

$$\begin{bmatrix} y_1(n) \\ y_2(n) \end{bmatrix} = \begin{bmatrix} w_{11}(n) & w_{12}(n) \\ w_{21}(n) & w_{22}(n) \end{bmatrix} * \begin{bmatrix} x_1(n) \\ x_2(n) \end{bmatrix} \tag{2.4}$$

or simply as,

$$\mathbf{y}(n) = \mathbf{W}(n) * \mathbf{x}(n), \tag{2.5}$$

where $\mathbf{y}(n) = [y_1(n)\ y_2(n)]^T$ is the vector of output signals approximating the original sources, and $\mathbf{W}(n)$ is the 2×2 un-mixing filter matrix whose elements are w_{ji}. The "permutation" designation is to express the fact that either permutation $\mathbf{y}(n) = [y_1(n)\ y_2(n)]^T$ or $\mathbf{y}(n) = [y_2(n)\ y_1(n)]^T$ is an acceptable result. The overall block diagram is given in Fig. 2.

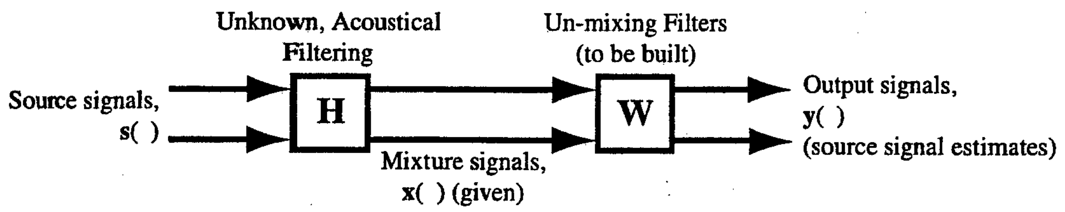

Unknown, Acoustical Filtering
Un-mixing Filters (to be built)
Source signals, s( )
H
W
Output signals, y( ) (source signal estimates)
Mixture signals, x( ) (given)

Figure 2: Block diagram of Blind Speech Separation setting.

3 Methods, Assumptions, and Procedures

3.1 Introduction

To date, one of the most successful methods for the blind separation of speech signals in reverberant environments is that recently proposed by Parra and Spence and further improved by Ikram and Morgan [3, 4]. Both of these algorithms have their roots in the work by Cardoso and are based on Frequency-Domain, Second Order Statistics (FDSOS). Here we attempt to decorrelate the speech signals from one another in the frequency domain, i.e. diagonalize covariance matrices. Since the original speech signals are nonstationary and uncorrelated but not independent, decorrelation is enough to theoretically, produce separated sources. The methods are frequency-domain-based so that a difficult time-domain deconvolution involving the acoustical filters can be avoided.

3.2 Frequency-Domain, Second Order Statistics-Based Algorithms

We first begin by transforming the time-domain convolutive mixture vector, $\mathbf{x}(n)$ in (2.3) to a frequency-domain, instantaneous mixture by computing its T-point short-time Fourier transform (STFT)

$$\begin{bmatrix} x_1(\omega,m) \\ x_2(\omega,m) \end{bmatrix} = \begin{bmatrix} H_{11}(\omega) & H_{12}(\omega) \\ H_{21}(\omega) & H_{22}(\omega) \end{bmatrix} \begin{bmatrix} s_1(\omega,m) \\ s_2(\omega,m) \end{bmatrix} \tag{3.1}$$

or

$$\mathbf{x}(\omega,m) = \mathbf{H}(\omega)\mathbf{s}(\omega,m) \tag{3.2}$$

where $\mathbf{x}(\omega,m) = [x_1(\omega,m)\ x_2(\omega,m)]^T$ is the STFT of the mixture signal vector, $\mathbf{s}(\omega,m) = [s_1(\omega,m)\ s_2(\omega,m)]^T$ is the STFT of the source signal vectors, $\mathbf{H}(\omega)$ is the matrix of acoustical frequency responses with $H_{ji}(\omega)$ the response from source i to microphone j , m is the block index, and ideally, $T = 2P$. For a given set of received data, $\mathbf{x}(n)$, $n = 0,\ldots,N-1$, we obtain the STFT as

$$\mathbf{x}(\omega,m) = \sum_{\tau=0}^{T-1} w(\tau)\mathbf{x}(\beta Tm+\tau)e^{-j2\pi\omega\tau/T}, \tag{3.3}$$

for $\omega = 1,\ldots,T$ and $m = 0,\ldots,N/(\beta T)-1$, where $w(\tau)$ is a window function and β $(0 < \beta \leq 1)$ is the data overlap factor. Fig. 3 illustrates the block structure of STFT and associated notation. Similarly (2.3) transforms to

$$\begin{bmatrix} y_1(\omega,m) \\ y_2(\omega,m) \end{bmatrix} = \begin{bmatrix} W_{11}(\omega) & W_{12}(\omega) \\ W_{21}(\omega) & W_{22}(\omega) \end{bmatrix} \begin{bmatrix} x_1(\omega,m) \\ x_2(\omega,m) \end{bmatrix} \tag{3.4}$$

or

$$\mathbf{y}(\omega,m) = \mathbf{W}(\omega)\mathbf{x}(\omega,m) \tag{3.5}$$

where $\mathbf{y}(\omega,m) = [y_1(\omega,m)\ y_2(\omega,m)]^T$ is the STFT of the output signal vector and $\mathbf{W}(\omega)$ is the matrix of un-mixing filter frequency responses with $W_{ji}(\omega)$ the response of the jith

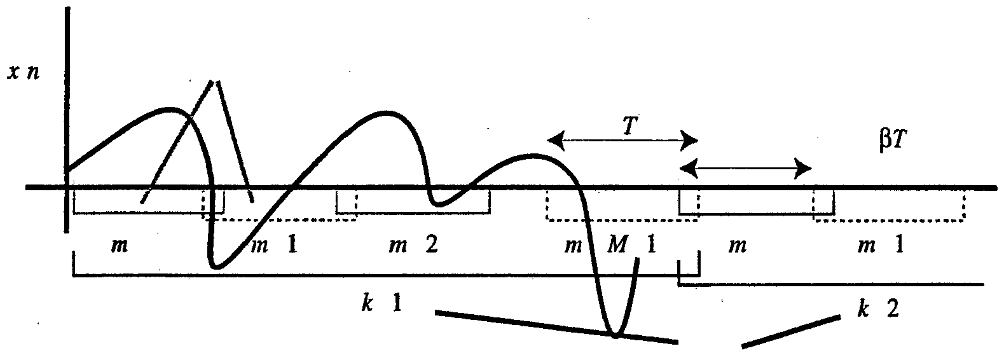


Figure 3: Block structure of STFT.

un-mixing filter. We employee the overlap-add method [5] to synthesize the separated time-domain speech signals as

$$\mathbf{y}(n) = \sum_{m=0}^{U-1} \tilde{\mathbf{y}}(n - m\beta T, m), \quad n = 0, \ldots, N-1 \tag{3.6}$$

where

$$\tilde{\mathbf{y}}(\tau, m) = \frac{1}{T} \sum_{\omega=0}^{T-1} \mathbf{y}(\omega, m) e^{j2\pi\omega\tau/T}, \quad \tau = 0, \ldots, T-1 \tag{3.7}$$

is the inverse STFT. In the synthesis given by (3.6), we have windowed blocks spaced by βT samples in time and overlapped and added to produce the output signals.

We assume that the source signals are quasi-stationary, i.e. statistics are slowly varying with time. Thus, we can treat the speech signals as a succession of superblocks of stationary signals, and analyze their spectrum on a time-varying basis. The assumption of uncorrelatedness of source signals $s_1(n)$ and $s_2(n)$ can therefore be represented in the time-domain by a cross-correlation of zero or in the frequency-domain by a Cross-Power Spectral Density (CPSD) of zero. The Power Spectral Density (PSD) matrix of source vector, $\mathbf{s}$ at frequency ω will then have following diagonal form

$$\begin{aligned} \mathbf{R_s}(\omega, k) &= E\left[\mathbf{s}(\omega, m)\mathbf{s}^H(\omega, m)\right] \\ &= \begin{bmatrix} \times & 0 \\ 0 & \times \end{bmatrix} \end{aligned} \tag{3.8}$$

where $\times$ is some non-zero value representing either the PSD of $s_1(n)$ or $s_2(n)$ and k is the superblock index.

Under the assumption of uncorrelated source signals, we seek to build frequency by frequency, an un-mixing filter matrix, $\mathbf{W}(\omega)$ that decorrelates the output signals $y_1(n)$ and

$y_2(n)$ at each frequency ω. Thus when applied to $\mathbf{x}(\omega, k)$, we have a diagonal PSD matrix for the output signals given by

$$\begin{aligned}\mathbf{R}_\mathbf{y}(\omega,k) &= E\left[\mathbf{y}(\omega,m)\mathbf{y}^H(\omega,m)\right] \\ &= \mathbf{W}(\omega)E\left[\mathbf{x}(\omega,m)\mathbf{x}^H(\omega,m)\right]\mathbf{W}^H(\omega) \\ &= \mathbf{W}(\omega)\mathbf{R}_\mathbf{x}(\omega,k)\mathbf{W}^H(\omega).\end{aligned} \tag{3.9}$$

Here we consider $\mathbf{y}(n)$ as the output of a linear shift invariant (LSI) system, $\mathbf{W}(n)$ with quasi-stationary input signals, $\mathbf{s}(n)$. The frequency-domain, covariance matrix, $\mathbf{R}_\mathbf{x}(\omega, k)$, can be estimated (also assuming ergodicity in each superblock) with an average of the outer products of the received signal vectors

$$\hat{\mathbf{R}}_\mathbf{x}(\omega,k) = \frac{1}{M}\sum_{m=0}^{M-1}\mathbf{x}(\omega,Mk+m)\mathbf{x}^H(\omega,Mk+m), \tag{3.10}$$

for $k = 0, \ldots, K-1$ provided the number of blocks in each superblock, M, is large enough. Note that in (3.10), the frequency-domain data is averaged over $M = N/(K\beta T)$, possibly overlapping, consecutive blocks to obtain the covariance at a super-block index k.

The decorrelation objective is therefore to find a sequence of un-mixing filter matrices, $\mathbf{W}(\omega)$ which minimizes the CPSD at each frequency or effectively diagonalizes (3.9) as much as possible for each frequency ω and each superblock k. The filters are then applied to $\mathbf{x}(\omega, m)$ resulting in (hopefully) decorrelated and thus separated output signals.

The second-order decorrelation criterion alone does not provide enough conditions to solve for $\mathbf{W}(\omega)$, unless the number of outputs is twice the number of inputs (four outputs for the two-input case) [3]. However, for non-stationary signals, we can write independent decorrelation equations for K sufficiently separated time intervals. As shown in [3], a set of K equations give a total of KT constraints on the $4Q$ unknown coefficients of $\mathbf{W}(\omega)$. In order to build an over-determined, least-squares problem (from which there will be at least one solution), it is required that $KT > 4Q$. The approximation of linear convolution (2.1) by circular convolution (3.1) requires $P \ll T$, and therefore $Q \ll T$. However, this constraint has to be relaxed. For example, in a typically sized room, P is always on the order of 10^3 at the sampling rate of 8kHz. But the estimation of the PSD as well as the computational complexity require that T not to be too large, typically, less than 10^4. In managing all these constraints, we always choose $Q \leq T$, and set $K \geq 4$ to avoid the under-determined case.

The un-mixing filter, $\mathbf{W}(\omega)$ for each frequency bin ω ($\omega = 1, \ldots, T$) that simultaneously satisfies the K decorrelation equations is obtained using an over-determined least-squares solution which seeks to minimize the off-diagonal elements of (3.9). In this framework, the un-mixing filter matrix is given by

$$\hat{\mathbf{W}}(\omega) = \arg\min_{\mathbf{W}(\omega)}\sum_{k=1}^{K}||\mathbf{V}(\omega,k)||_F^2, \tag{3.11}$$

where $||\cdot||_F^2$ is the Frobenius norm squared (sum of squares of all elements) and the off-diagonal matrix is given by

$$\mathbf{V}(\omega,k) = \mathbf{W}(\omega)\hat{\mathbf{R}}_\mathbf{x}\mathbf{W}^H(\omega) - \operatorname{diag}\left[\mathbf{W}(\omega)\hat{\mathbf{R}}_\mathbf{x}\mathbf{W}^\mathrm{H}(\omega)\right], \tag{3.12}$$

where diag $[\cdot]$ is the diagonal matrix formed by extracting the diagonal elements of the matrix argument. The least-squares solution to (3.11) can be searched for using the well-known steepest descent algorithm

$$\mathbf{W}^{(l+1)}(\omega) = \mathbf{W}^{(l)}(\omega) - \mu(\omega)\frac{\partial}{\partial \mathbf{W}^{(l)^H}(\omega)}\left\{\sum_{k=1}^{K}||\mathbf{V}^{(l)}(\omega,k)||_F^2\right\} \tag{3.13}$$

for $\omega = 1,\ldots,T$. In this equation, we iteratively update the filter matrix $\mathbf{W}(\omega)$ by adjusting it in the direction which leads to the "smallest" $\mathbf{V}(\omega,k)$, i.e. smallest off-diagonal values of (3.9). In Appendix A, we provide the complete derivation for the gradient term in (3.13). Substituting the result of the derivation into (3.13) leads to the filter update

$$\mathbf{W}^{(l+1)}(\omega) = \mathbf{W}^{(l)}(\omega) - \mu(\omega)\sum_{k=1}^{K}\mathbf{V}(\omega,k)\mathbf{W}(\omega,k)\hat{\mathbf{R}}_{\mathbf{x}}(\omega,k). \tag{3.14}$$

In [3], it is noted that the gradient terms scale with the square of the signal powers, which vary considerably across frequency and result in wild fluctuation of the gradient terms for different frequencies. Thus, we use a step size function

$$\mu(\omega) = \frac{\tilde{\mu}}{\sum_{k=1}^{K}||\hat{\mathbf{R}}_{\mathbf{x}}(\omega,k)||_F^2} \tag{3.15}$$

where $\tilde{\mu}$ is a normalized step size. To fix the arbitrary scaling in $\mathbf{W}(\omega)$, only the off-diagonal elements of $\mathbf{W}(\omega)$ are updated, thus diagonal elements always remain at their initial values.

3.3 The Permutation Inconsistency/Spectral Resolution Tradeoff

If $\mathbf{W}(\omega)$ diagonalizes $\hat{\mathbf{R}}_{\mathbf{x}}(\omega,k)$, i.e.

$$\hat{\mathbf{R}}_y(\omega,k) = \begin{bmatrix} D_{11} & 0 \\ 0 & D_{22}\end{bmatrix} \tag{3.16}$$

then $\mathbf{W}'(\omega) = \mathbf{P}\mathbf{W}(\omega)$, with $\mathbf{P}$ a permutation matrix, also diagonalizes $\hat{\mathbf{R}}_{\mathbf{x}}(\omega,k)$

$$\begin{aligned}\hat{\mathbf{R}}'_{\mathbf{y}}(\omega,k) &= \mathbf{P}\mathbf{W}(\omega)\hat{\mathbf{R}}_{\mathbf{x}}(\omega,k)\mathbf{W}^H(\omega)\mathbf{P}^T \\ &= \begin{bmatrix} 0 & 1 \\ 1 & 0\end{bmatrix}\begin{bmatrix} D_{11} & 0 \\ 0 & D_{22}\end{bmatrix}\begin{bmatrix} 0 & 1 \\ 1 & 0\end{bmatrix} \\ &= \begin{bmatrix} D_{22} & 0 \\ 0 & D_{11}\end{bmatrix}. \end{aligned} \tag{3.17}$$

Note that $\mathbf{W}(\omega)$ and its permutation $\mathbf{W}'(\omega)$ will lead to the same cost, $J(\omega)$ at each frequency ω. Therefore the least-squares solution may consist of a different permutation of $\mathbf{W}(\omega)$ at each frequency bin, causing arbitrary permutations in the separated signals over the spectrum. Fig. 4 illustrates the permutation problem. Here the spectral components

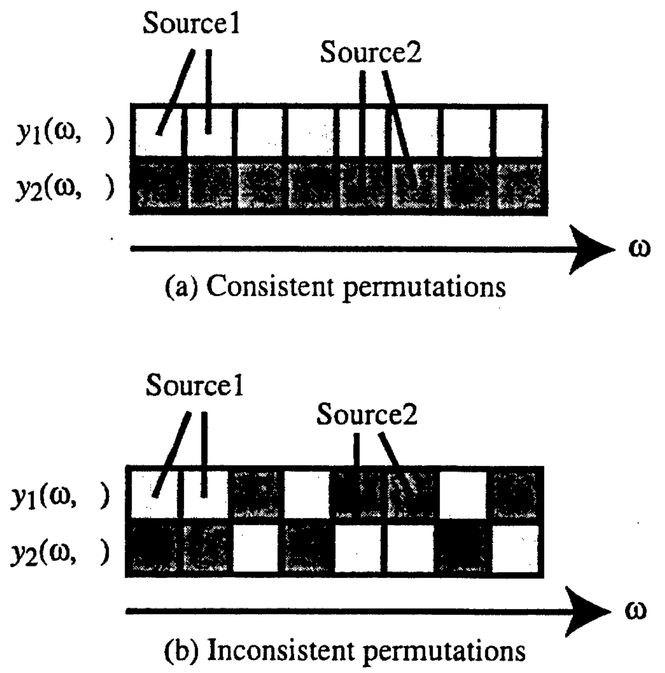


Figure 4: Permutations across the spectrum of the output signals.

while uncorrelated at each frequency, are not properly collected at the same channel output, i.e. mixed in the frequency-domain.

In [3], this problem is solved by constraining the length of $\mathbf{W}(n)$ to $Q \ll T$, thereby forcing the solutions to be continuous or "smooth" in the frequency-domain. This constraint is applied by starting the gradient algorithm with an initial value of $\mathbf{W}(\omega)$ that satisfies $\mathbf{W}(n) = 0$ for $Q \leq n \leq T-1$, and then following the constrained gradient, which is a projection of the gradient term in (3.13) with the projection operator

$$\mathbf{G} \;=\; \mathbf{F}\mathbf{Z}\mathbf{F}^{-1} \tag{3.18}$$

where $\mathbf{F}$ is the Discrete Fourier Transform (DFT) matrix and $\mathbf{Z}$ is the truncation matrix given by

$$Z_{ij} \;=\; \begin{cases} 1, & i=j,\ i<Q \\ 0, & i=j,\ i\geq Q \\ 0, & i\neq j. \end{cases} \tag{3.19}$$

The projection is implemented by first transforming the gradient to the time-domain, zeroing all components for $n > Q$, and transforming back to the frequency-domain. The main problem with this solution to the permutation problem is the limited size of un-mixing filters. It is well-known that good separation performance requires long filters to accurately model the inverse of the acoustical impulse response. In [3], it was proposed that a large Q (filter size) can be achieved by increasing the block size T. However, as described in the previous section, T cannot exceed 10^4 for a sampling rate of 8kHz. Thus, as shown in [4], Q is limited to a value on the order of 10^2 in order to minimize the permutation problem.

However, with this size of Q there is insufficient spectral resolution in frequency-domain. The permutation-inconsistency/spectral-resolution tradeoff is pictured in Fig. 5 with varying Q and fixed $T = 2048$. As one can see the best Signal-to-Interference Ratio Improvement (SIRI), formally defined in the next section, is 13dB and achieved with $Q = 400$.

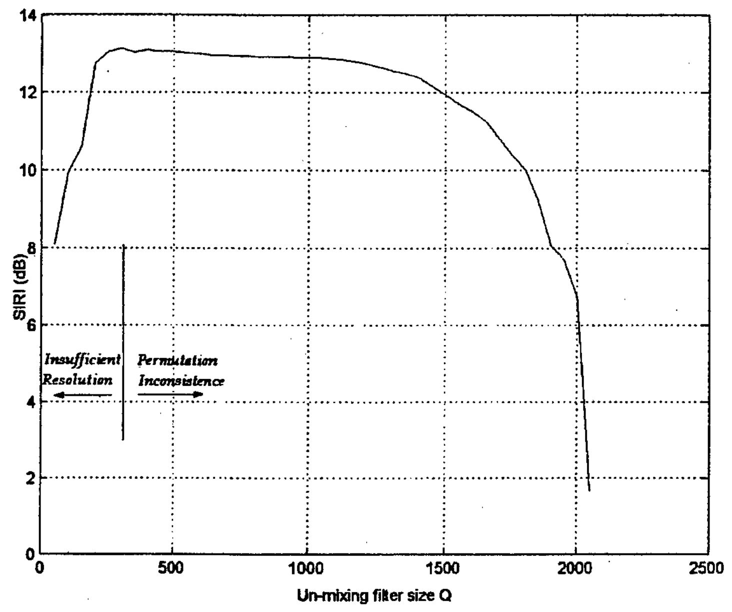


Figure 5: Average SIRI versus un-mixing filter length Q.

3.4 A Multiresolution Approach for FDADF

A multiresolution frequency-domain (MRFD) algorithm was proposed by Ikram and Morgan to satisfy the conflicting requirements of permutation alignment and spectral resolution [4]. In the multistage procedure, the FDSOS method of (3.14) is used with increasing values of filter length Q at each stage of the algorithm. Different values of Q imply varying the frequency-domain resolution of $\mathbf{W}(\omega)$ at each stage, hence the name "multiresolution." The rationale behind such an approach is to allow the permutations to align themselves using a smaller value of $Q << T$ in the early stages of the algorithm. Once the permutations are aligned, they tend to retain their order even if the value of Q is increased. The increase in the value of Q, however, provides the desired spectral resolution which is lacking in the early stages.

A block diagram illustrating the blind separation of speech signals using the MRFD algorithm is shown in Fig. 6. The mixing stage is followed by S un-mixing stages, each of which attempts to further separate the sources using an un-mixing filter with increased spectral resolution. Note that the separated output signals from each un-mixing stage (after convergence) at each stage are carried over as the initial weights for the following stage. To initiate the separation procedure, $\mathbf{W}_1^{(0)} = \mathbf{I}_2$ is used in the first stage. Since the un-mixing of the speech signals is carried out by S different sets of weights, the SIR_o for the MRFD algorithm can be computed using the overall multichannel response

$$\mathbf{A}(\omega) \;=\; \mathbf{W}_s(\omega)\mathbf{W}_{s-1}(\omega)\ldots\mathbf{W}_1(\omega)\mathbf{H}(\omega) \qquad (3.20)$$

in place of $\mathbf{H}(\omega)$ in (3.24).

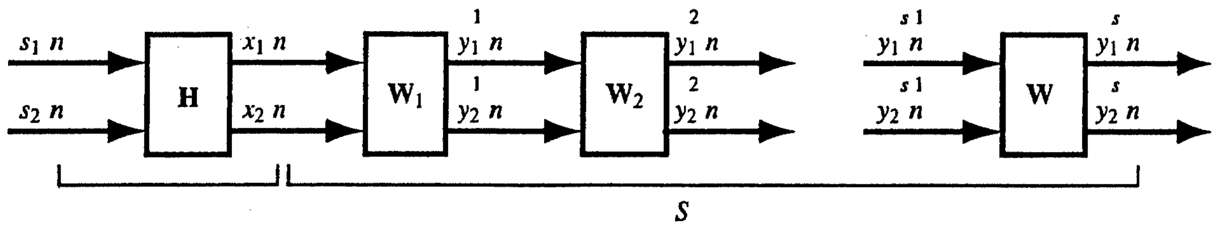


Figure 6: Block diagram of Multiresolution Frequency Domain blind speech separator.

It is well-known that a blind estimate of $\mathbf{W}(\omega)$ at frequency ω can be best obtained up to a scale and a permutation. Therefore, at each frequency ω, the separated signal $s_1(\omega)$ may have $\hat{s}_1(\omega) = \gamma s_1(\omega)$ or $\hat{s}_1(\omega) = \gamma s_2(\omega)$, where γ is an arbitrary scaling factor, and the second possibility arises from a simple interchange of the rows of the un-mixing filter matrix $\mathbf{W}(\omega)$. Consequently, the recovered source signal, $\hat{s}_i$ is not necessarily a consistent estimate of s_i over all frequencies. What occurs is that each "separated" output contains scaled, spectral energy at particular frequencies from the other source and vice versa. In [3], this problem was solved by constraining the length of $\mathbf{W}(\omega)$ to $Q < T$, thereby forcing the solution to be smooth or continuous in the frequency domain. However, due to insufficient spectral resolution, actual SIRI is modest.

In [6], the authors explore SIRI for various choices of Q. As mentioned above, for $Q < T$, insufficient spectral resolution limits actual SIRI. Although choosing $Q \approx T$ provides sufficient spectral resolution, sufficient continuity of the un-mixing filters in the frequency domain is not achieved. A sub-optimal choice of Q, however, can be found which provides a good compromise between the two competing objectives although performance is still modest and below theoretical SIRI when $Q \approx T$ and permutations of $\mathbf{W}(\omega)$ are perfectly aligned.

Experimental results indicate good performance with only two stages where Q_2 is approximately equal to the reverberation time of the room in sample periods and (as a rule of thumb), $Q_1 \approx Q_2/4$. A typical reverberation time of 250ms at an 8kHz sampling rate would give $Q_2 = 2000$. When rounded up to a power of two for a convenient choice of block length and FFT implementation, we have $Q_2 = 2048$.

3.5 Modifications to MRFD

In this section, we discuss several modifications to the MRFD algorithm based on our own independent research which improve computational efficiency and performance. Results with using the modified MRFD algorithm are compared to the original algorithm in the next section.

3.5.1 Real-Valued Constraint on the Un-mixing Filters

The coefficients of the acoustical impulse responses, $\mathbf{H}(n)$ will always be real-valued numbers and thus the un-mixing filter coefficients, $\mathbf{W}(n)$ will also real-valued. Exploiting this fact can reduce the computational complexity by half since from the the Hermitian property of the DFT, we have

$$\mathbf{W}(\omega) \;=\; \mathbf{W}^*(Q-\omega) \tag{3.21}$$

for $\omega = 0,\ldots,Q-1$. Therefore we need only solve the least squares problem in (3.11) for $2(Q+1)$ unknown parameters given $K(T+1)/2$ constraints in (3.9).

3.5.2 Reinitialization of $\mathbf{W}(\omega)$ at Each Stage

In [4], the un-mixing filters $\mathbf{W}_q(\omega)$ are initialized at each stage with the converged coefficients of $\mathbf{W}_{q-1}^{(L)}(\omega)$ from the previous stage—a technique known as "cascaded initialization". For the first stage, we set $\mathbf{W}_1^{(0)}(\omega) = \mathbf{I}$. However, the algorithm is not directly optimizing the un-mixing filters, but instead minimizing the CPSDs. Also, the gradient algorithm may reach a local minima at interim stages. Therefore, reinitializing $\mathbf{W}_q^{(0)}(\omega)$ may help the algorithm to reach different local minima at the qth stage and "save" the improvement achieved in current stage. In order to evaluate the objective function $J(\omega)$ at each iteration, we measure the diagonalization of $\hat{\mathbf{R}}_{\mathbf{y}}(\omega,k)$ with

$$O \;=\; \frac{\sum\limits_{k=1}^{4}\sum\limits_{\omega=0}^{T-1}\sum\limits_{i=1}^{2}|\hat{R}_{y_{ii}}(\omega,k)|^2}{\sum\limits_{k=1}^{4}\sum\limits_{\omega=0}^{T-1}\sum\limits_{i=1}^{2}\sum\limits_{j=1,j\neq i}^{2}|\hat{R}_{y_{ij}}(\omega,k)|^2} \tag{3.22}$$

where $\hat{R}_{y_{ij}}(\omega,k)$ is the ijth element of the PSD matrix of output defined in (3.9).

Fig. 7 compares the learning curves of the cascaded initializing strategy (two dash lines with +) [4] and those of the reinitialization approach (two solid lines) with the measurement in (3.22) for the two-stage MRFD. The lower lines (solid line and dashed line are overlapped) are the learning curves for the first stage, and the upper lines are the learning curves for the second stage. We can see in the case of reinitialization O is "saved" at the end of the first stage and is continuously improved at the second stage. The "save-attack" behavior is thought to be due to the update of $\hat{\mathbf{R}}_{\mathbf{x}}(\omega,k)$ at the beginning of each stage. A quantitative comparison of SIRI is also given in Table 3.

3.5.3 Projection of the Un-mixing Filters $\mathbf{W}(\omega)$

As explicitly stated in [7], the projection $\mathbf{G}$ (3.18) is implemented by transforming the gradient into the time-domain, zeroing all components with $\tau > Q$, and transforming back to the frequency domain. Before truncating the gradient in the time-domain, we have to first normalize it by $\sum_{k=1}^{K}||\hat{\mathbf{R}}_{\mathbf{x}}(\omega,k)||_F^2$, in order to stabilize the algorithm. For simplicity in the Matlab implementation of the MRFD algorithm, we instead apply the projection operator $\mathbf{P}$ directly to the un-mixing filters $\mathbf{W}(\omega)$ after each update. Our results show that both

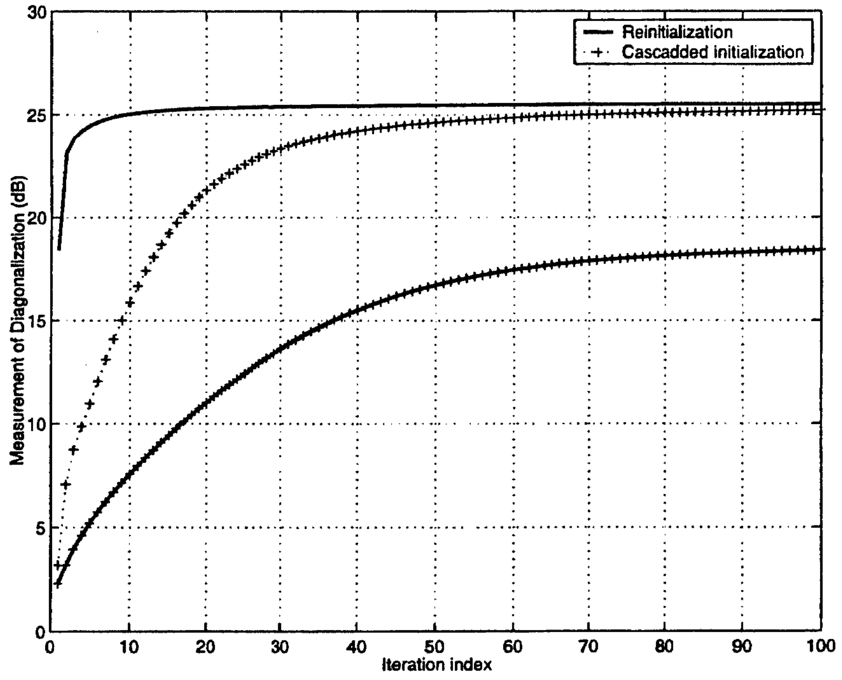


Figure 7: Comparison of two initializing strategies.

projection methods achieve the same SIRI performance due to the linearity property of the DFT.

We summarize the MRFD algorithm with the above modifications in Fig. 8.

3.6 Performance Metrics

Researchers evaluate the performance of blind speech separation and speech enhancement algorithms in various ways. These include speech recognition rates [3]; plots of separated signals [8], [9], [10]; and analysis of the improvement to the Signal-to-Interference Ratio after processing [3], [4]. In this work, we have exclusively measured algorithm performance using the SIRI since this is most easily computed measure and allows easy comparison with other algorithms. For further information on how improvements to the SIR impacts automatic speech recognition (ASR) see [11].

The average input signal-to-interference ratio (SIR_i) is defined as the ratio of the total signal power obtained via direct channels (h_{11}, h_{22}) to the total signal power obtained via cross channels (h_{21}, h_{12}), i.e.

$$\text{SIRi} \;=\; \frac{\sum\limits_{\omega=0}^{T-1}\sum\limits_{i=1}^{2}|H_{ii}(\omega)|^2\,|s_i(\omega)|^2}{\sum\limits_{\omega=0}^{T-1}\sum\limits_{i=1}^{2}\sum\limits_{j=1,j\neq i}^{2}|H_{ij}(\omega)|^2\,|s_j(\omega)|^2}. \qquad (3.23)$$

Replacing $\mathbf{H}(\omega)$ by $\mathbf{W}(\omega)\mathbf{H}(\omega)$ similarly defines the average post-processed, or output signal-

```
$\mathbf{x}(\omega,k)=\sum_{\tau=0}^{T-1} w(\tau)\mathbf{x}(m\beta T+\tau)e^{-j2\pi\omega\tau/T}$; Short-Time Fourier Transform
For $q=1:SG$
  $\hat{\mathbf{R}}_{\mathbf{x}}(\omega,k)=\frac{1}{M}\sum_{m=0}^{M-1}\mathbf{x}(\omega,Mk+m)\mathbf{x}^H(\omega,Mk+m)$; Estimate PSD
  $\mathbf{V}(\omega,k)=\text{Off}\left[\hat{\mathbf{R}}_{\mathbf{x}}(\omega,k)\right]$; Build cost matrix
  $\mu(\omega)=\frac{\mu}{\sum_{k=1}^{K}\|\hat{\mathbf{R}}_{\mathbf{x}}(\omega,k)\|_F^2}$; Normalize step size
  For $l=1:L$
    $\mathbf{C}=2\sum_{k=1}^{K}\text{Off}[\mathbf{V}(\omega,k)\mathbf{W}^{(l)}(\omega)\hat{\mathbf{R}}_{\mathbf{x}}(\omega,k)]$; Build correction matrix
    $\mathbf{W}^{(l+1)}(\omega)=\mathbf{W}^{(l)}(\omega)-\mu(\omega)\mathbf{C}$; Update $W(\omega)$ at each frequency
    $\mathbf{W}^{(l+1)}(T-\omega)=[\mathbf{W}^{(l+1)}(\omega)]^*$; Apply symmetric constraint
    $\mathbf{G}=\mathbf{F}\mathbf{Z}_q\mathbf{F}^{-1}$; Projection operator
    $W_{ij}^{(l+1)}=\mathbf{G}W_{ij}^{(l+1)}$; Apply length constraint on un-mixing filters
  End
  $\mathbf{x}^{(q+1)}(\omega,m)=\mathbf{W}(\omega)\mathbf{x}^{(q)}(\omega,m)$; Current stage separation
End
$\hat{\mathbf{y}}(\tau,m)=\frac{1}{T}\sum_{\omega=0}^{T-1}\mathbf{x}^{(SG)}(\omega,m)e^{j2\pi\omega\tau/T}$
$\mathbf{y}(n)=\sum_{m=0}^{U-1}\hat{\mathbf{y}}(n-m\beta T,m)$; Synthesize time-domain signals
```

Figure 8: Modified MRFD Algorithm for speech separation.

to-interference ratio (SIR_o).

$$\text{SIRo} \;=\; \frac{\sum_{\omega=0}^{T-1}\sum_{i=1}^{2}|[\mathbf{W}(\omega)\mathbf{H}(\omega)]_{ii}|^2\,|s_i(\omega)|^2}{\sum_{\omega=0}^{T-1}\sum_{i=1}^{2}\sum_{j=1,j\neq i}^{2}\left|[\mathbf{W}(\omega)\mathbf{H}(\omega)]_{ij}\right|^2|s_j(\omega)|^2} \qquad (3.24)$$

The objective of blind speech separation algorithms is to obtain a high SIR improvement (SIRI) given by the ratio of (3.23) to (3.25)

$$\text{SIRI} \;=\; \text{SIR}_\text{o}/\text{SIR}_\text{i}. \qquad (3.25)$$

In [12], Schobben *et al.* divided the suite of test cases into two main categories:

1. *Controllable Synthetic Separation Problems*
In this case, the channel impulse responses and source signals are known a prior due to the simulated approach to testing. In our experiments, we simulate the impulse responses from an acoustic source to multiple microphones using the image method described in [13]. The simulated room is shown in Fig. 9. The image method requires the dimensions of the virtual room; coordinates of each source, $(x_{s_i}, y_{s_i}, z_{s_i})$; coordinates of each microphone, $(x_{m_i}, y_{m_i}, z_{m_i})$; and reflection coefficients of the six walls, ρ_l, $l=1,\ldots,6$. Parameters listed in Table 1 are used in our simulations except otherwise specified. For testing purposes,

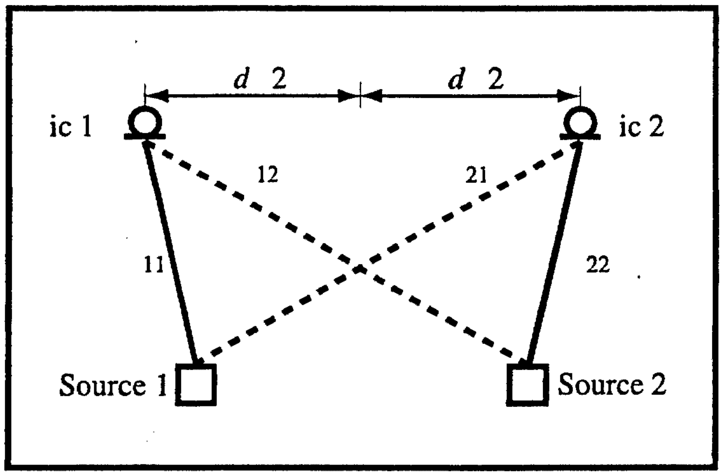


Figure 9: Room geometry (coordinate units in meter).

speech mixtures were digitally synthesized according to (2.1) using speech sources drawn from the Linguistic Data Consortium's (LDC's) Texas Instruments, Massachusetts Institute of Technology (TIMIT) speech corpus and filtered with impulses simulated with the image method. This speech corpus consists of short utterances from a large sample of American English speakers. For more information on this speech database see [14].

Table 1: Simulated room parameters.

Parameters	**Values** (in meters)
Source 1 coordinates	(2.13, 0.91, 1.52)
Source 2 coordinates	(3.35, 0.91, 1.52)
Microphone 1 coordinates	(2.23, 2.74, 1.68)
Microphone 2 coordinates	(2.83, 2.74, 1.68)
Room dimensions	(5.06, 3.41, 2.44)
Reflection coefficient, ρ	(0.3, 0.3, 0.3, 0.3, 0.3, 0.3)

2.*Real-World Recording Problems*
Here, original source signals and impulse responses are not available. However, we can still estimate direct-channel signal power (numerator) and cross-channel signal power (denominator) in (3.23) by using alternating source signals [3]. We estimate the contributions of source i while source i is "on" and all other sources are "off". For a two-channel mix, Fig. 10 shows this approach. In this case, we have four mixed signals and four separated signals, including the direct-channel contributions $(x_{11}, x_{22}, y_{11}, y_{22})$ and the cross-channel contributions $(x_{12}, x_{21}, y_{12}, y_{21})$. We can measure the SIRI as in (3.25) where, in this approach, we

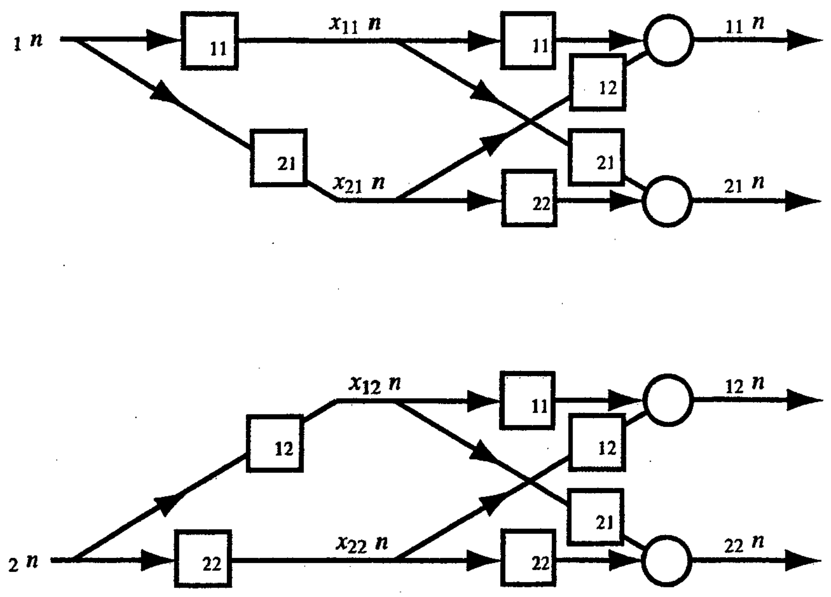


Figure 10: Block diagram of indirect SIRI metric.

have

$$\text{SIRi} = \frac{\sum_{\omega=0}^{T-1}\sum_{i=1}^{2}|x_{ii}(\omega)|^2}{\sum_{\omega=0}^{T-1}\sum_{i=1}^{2}\sum_{j=1,j\neq i}^{2}|x_{ij}(\omega)|^2} \tag{3.26}$$

and

$$\text{SIRo} = \frac{\sum_{\omega=0}^{T-1}\sum_{i=1}^{2}|y_{ii}(\omega)|^2}{\sum_{\omega=0}^{T-1}\sum_{i=1}^{2}\sum_{j=1,j\neq i}^{2}|y_{ij}(\omega)|^2}. \tag{3.27}$$

For real-world testing, we used speech mixtures drawn from Lee's recordings available over the Internet at

`http://medi.uni-oldenburg.de/members/ane/pub/demo_asa99/index.html`

and the NMSU BSS speech corpus described below.

3.7 Modified MRFD Implementation

The speech separation code contains eight different .c files and one .h (header) file. The first and main file is the speech.c. The program needs the standard stdio.h, stdlib.h and math.h to compile correctly. Several of the lower-level math and signal processing routines can be found in [15].

- `speech.c` – Contains the `main()` function and is where variable initialization, memory allocation, and calls to other functions take place. This is the basis from where the algorithm is processed.
- `complex.c` – This file contains all the math functions for complex arithmetic as well as defines the structure for complex numbers. The functions that are defined are as follows.
  - `cmplx()` – Converts real numbers to complex
  - `conjg()` – Conjugate of complex number
  - `cadd()` – Complex addition
  - `csub()` – Complex subtraction
  - `cmul()` – Complex multiplication
  - `rmul()` – Multiplication by real
  - `cdiv()` – Complex division
  - `rdiv()` – Division by real
  - `real()` – Real part of complex number
  - `aimag()` – Imaginary part of complex number
  - `cexp()` – Complex Exponential
- `cmplx.h` – Defines of all the complex functions
- `fft.c` – In-place decimation-in-time (DIT) Fast Fourier Transform (FFT)
- `ifft.c` – Inverse Fast Fourier Transform (IFFT)
- `bitrev.c` – Does bit reversing of a B-bit integer, used by the Fourier Transforms
- `dftmerge.c` – Discrete Fourier Transform (DFT) for radix-2 DIT FFT
- `shuffle.c` – In-place shuffling (bit-reversal) of a complex array
- `swap.c` – Swap two complex numbers

The change of the C implementation from a non-real-time model to a real-time model would require a few foreseeable changes in the code. There are a couple loops that operate on the entire signal, the normalization for example. The parameters for these loops will need to be modified for a real-time implementation. Some loops will need to be eliminated as well and replaced with control code for which block of samples it is operating on. The control mechanisms for the loading and operating of samples in buffer blocks will also need to be added. For a real-time implementation the code would most likely need to be optimized as

Table 2: Execution times for processing 50s long speech signals using C-implementation of Modified MRFD.

real	0m20.551s
user	0m19.980s
sys	0m0.420s

well, there are many places in the non-real-time model where this can be done. In general the core algorithm in the C code implementation is sound and would most likely only require other code to be wrapped around it.

Benchmarks for processing two 50s long speech signals (8kHz sample rate) are listed in Table 2. Processing times were determined with the following instruction

```
time ./speech mix1.txt mix2.txt
```

The test PC has dual 1.2GHz AMD Athlon MP Processors, 1GB of DDR RAM, Red Hat Linux 7.2 (Kernel 2.4.16). The codes were compiled under gcc-2.96 with glibc-2.2.4. We note that the program is not multithreaded so it has no benefit of a dual processor machine. Code is compiled with Athlon optimizations.

3.8 NMSU BSS Speech Corpus

In this work, we developed a speech corpus in order to standardize evaluation and testing of this and future speech separation/speech enhancement algorithms. This corpus is available (free of charge) over the Internet at

```
http://www.ece.nmsu.edu/~pdeleon/BSS/index.html
```

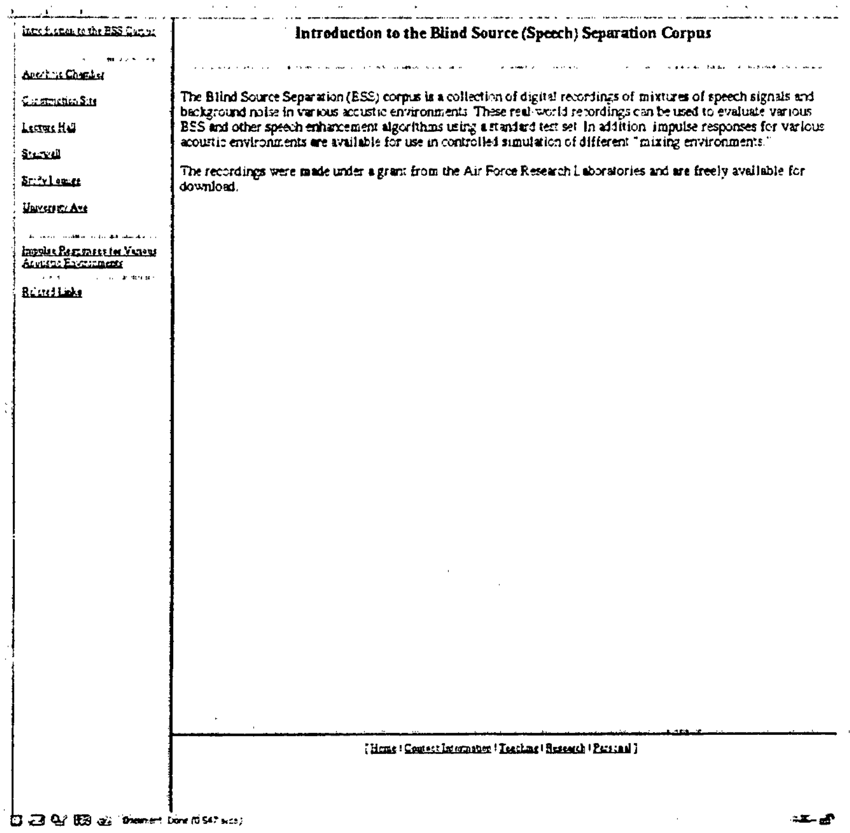
The corpus contains a collection of impulse responses and speech recordings from various environments and is more extensive than other available databases specifically targeted at BSS. Also included on the website are more detailed descriptions of recording procedure, environment, speakers, etc.... Figs. 11 and 12 show sample screen shots of the web pages for the NMSU BSS speech corpus. Here we briefly described collection procedure and available recordings.

3.8.1 Impulse Responses for Various Acoustic Environments

In order to provide researchers the ability to synthesize signals in various acoustic environments, impulse responses were measured for various environments. For each environment, a pair of direct channel impulse responses (left source to left microphone and right source to right microphone) and a pair of cross channel impulse responses (left source to right microphone and left source to right microphone) were measured as in Fig. 13. The impulse responses are only accurate to within a delay since the stimulus used to excite the room response was not synchronized to the recorder. In order to simplify, all responses begin 50ms before the main peak (direct path) and extend a total of 3sec. Typically, the cross channel responses have a longer delay than the direct channel since the path is longer.

Recording Equipment Used in All Impulse Response Measurements

The following recording equipment was used in collection of impulse responses.



Introduction to the Blind Source (Speech) Separation Corpus

The Blind Source Separation (BSS) corpus is a collection of digital recordings of mixtures of speech signals and background noise in various acoustic environments. These real-world recordings can be used to evaluate various BSS and other speech enhancement algorithms using a standard test set. In addition, impulse responses for various acoustic environments are available for use in controlled simulation of different "mixing environments."

The recordings were made under a grant from the Air Force Research Laboratories and are freely available for download.

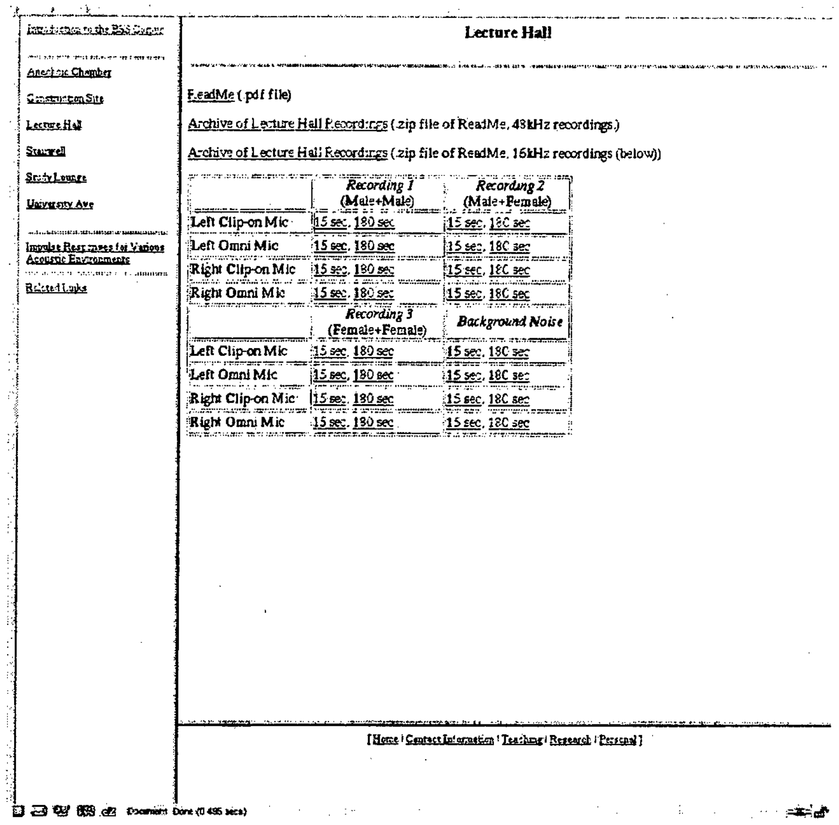
[Home | Contact Information | Teaching | Research | Personal]
Document: Done (0.547 secs)

Figure 11: Main web page for the NMSU Blind Speech Corpus.

- (1) Dell Dimension XPS R400 PC (Pentium II @ 400MHz, 128MB RAM, 40GB disk, Windows 98) (www.dell.com)
- (1) Echo Layla 20-bit multitrack recording system (www.)
- (2) Shure omni-directional microphone Model VP64A2 (www.shure.com)
- (2) Applied Research And Technology (ART) Professional processor series tube preamp (www.art.com)
- (2) Balanced (XLR) microphone cables from microphones to preamps
- (2) Balanced (XLR) microphone cables from preamps to Layla

Recording/Editing Software
The following software was used in collection of impulse responses.

- Syntrillium Software Corporation's Cool Edit Pro v1.2 for recording and (www.syntrillium.com)
- Aurora Convolve with Clipboard and Generate Log Sweep plug-ins for Cool Edit pro for impulse response measurements (www.ramsete.com/aurora/)
- SoundApp PPC v2.7.3 for sample rate conversion to 16kHz from 48kHz (www.download.com)



Introduction to the BSS Corpus

Anechoic Chamber

Construction Site

Lecture Hall

Stairwell

Study Lounge

University Ave

Impulse Responses for Various Acoustic Environments

Related Links

Lecture Hall

ReadMe (.pdf file)

Archive of Lecture Hall Recordings (.zip file of ReadMe, 48kHz recordings)

Archive of Lecture Hall Recordings (.zip file of ReadMe, 16kHz recordings (below))

	Recording 1 (Male+Male)	*Recording 2* (Male+Female)
Left Clip-on Mic	15 sec, 180 sec	15 sec, 180 sec
Left Omni Mic	15 sec, 180 sec	15 sec, 180 sec
Right Clip-on Mic	15 sec, 180 sec	15 sec, 180 sec
Right Omni Mic	15 sec, 180 sec	15 sec, 180 sec
	Recording 3 (Female+Female)	*Background Noise*
Left Clip-on Mic	15 sec, 180 sec	15 sec, 180 sec
Left Omni Mic	15 sec, 180 sec	15 sec, 180 sec
Right Clip-on Mic	15 sec, 180 sec	15 sec, 180 sec
Right Omni Mic	15 sec, 180 sec	15 sec, 180 sec

[Home | Contact Information | Teaching | Research | Personal]

Document: Done (0.495 secs)

Figure 12: Sample web page for the the lecture hall recording environment.

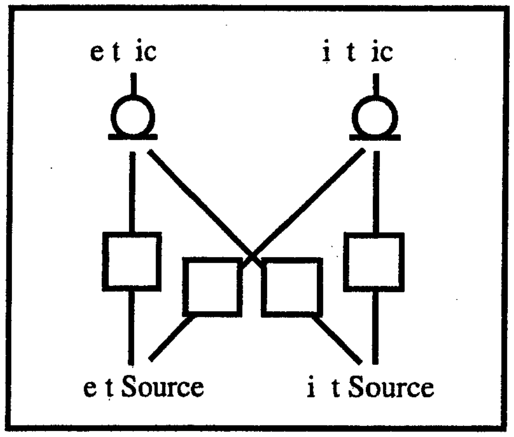


Figure 13: Direct and cross channel impulse responses.

Room Impulse Response Measurement

The Aurora plug-in for Cool Edit Pro utilizes a Chirp (sinusoidal sweep) stimulus in order to compute the room impulse response which is sampled at 48kHz. Impulse responses are also provided at a 16kHz sample rate.

Anechoic Chamber

The anechoic chamber is located on the New Mexico State University campus in the Klipsch School of Electrical and Computer Engineering (Thomas and Brown, Room 311). Fig. 14 shows the schematic of recording setup with dimensions. The available impulse response files

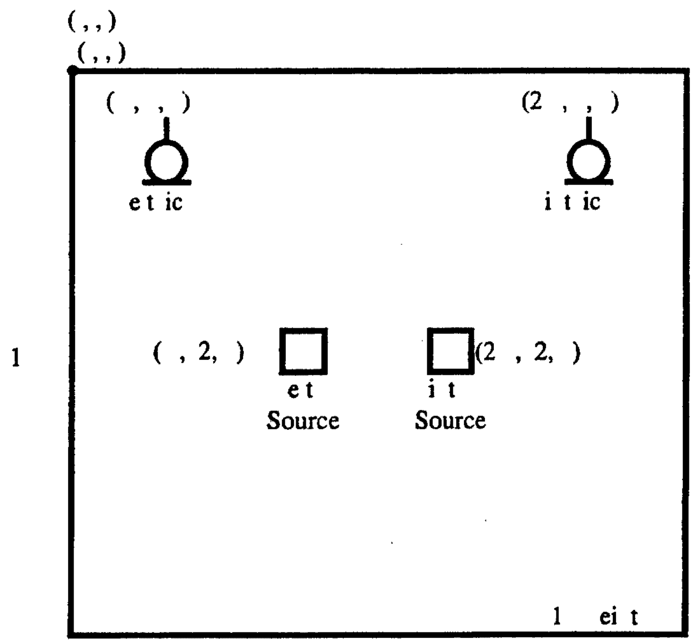


Figure 14: Schematic of Anechoic Chamber recording setup.

are:

Anechoic_ImpResp_16k_LL.wav	Anechoic_ImpResp_16k_LR.wav
Anechoic_ImpResp_16k_RL.wav	Anechoic_ImpResp_16k_RR.wav
Anechoic_ImpResp_48k_LL.wav	Anechoic_ImpResp_48k_LR.wav
Anechoic_ImpResp_48k_RL.wav	Anechoic_ImpResp_48k_RR.wav

Bathroom

The men's bathroom is located on the New Mexico State University campus in the Klipsch School of Electrical and Computer Engineering (Goddard Hall, Room 101). Fig. 15 shows the schematic of recording setup with dimensions. The available impulse response files are:

Bathroom_ImpResp_16k_LL.wav	Bathroom_ImpResp_16k_LR.wav
Bathroom_ImpResp_16k_RL.wav	Bathroom_ImpResp_16k_RR.wav
Bathroom_ImpResp_48k_LL.wav	Bathroom_ImpResp_48k_LR.wav
Bathroom_ImpResp_48k_RL.wav	Bathroom_ImpResp_48k_RR.wav

Large Room (Conference Room)

The large room is located on the New Mexico State University campus in the Klipsch School of Electrical and Computer Engineering (Goddard Hall, Room 167). Fig. 16 shows the schematic of recording setup with dimensions. The available impulse response files are

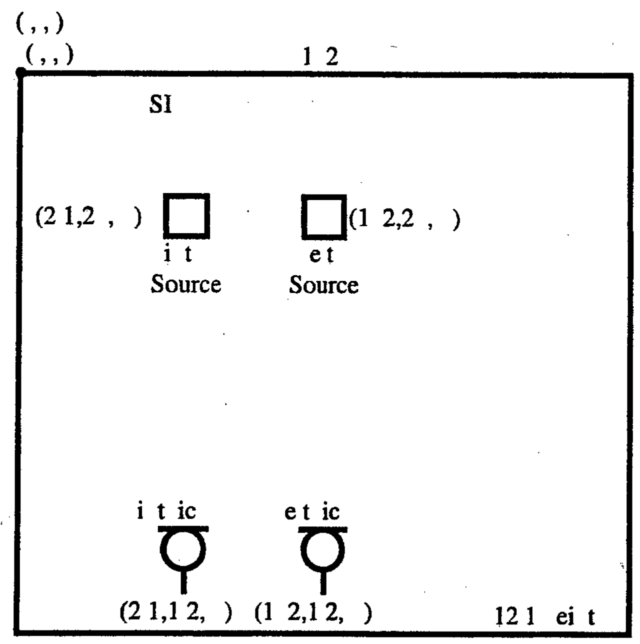


Figure 15: Schematic of Bathroom recording setup.

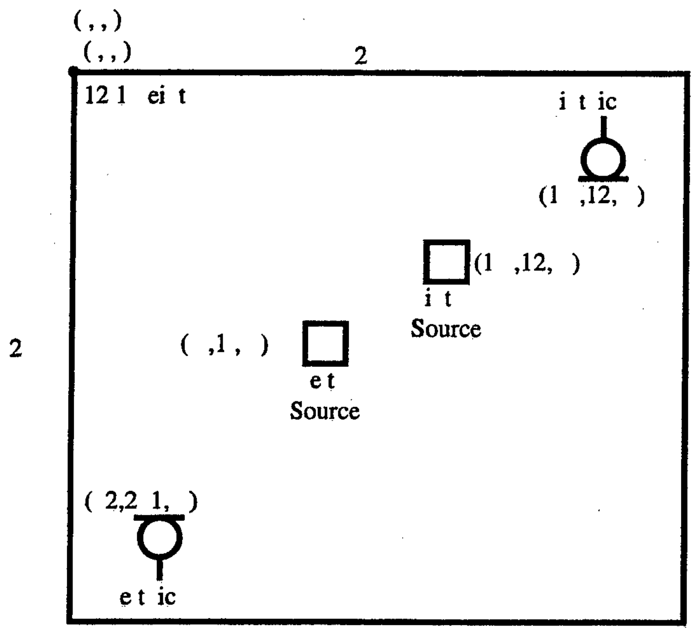


Figure 16: Schematic of Large Room recording setup.

LargeRoom_ImpResp_16k_LL.wav	LargeRoom_ImpResp_16k_LR.wav
LargeRoom_ImpResp_16k_RL.wav	LargeRoom_ImpResp_16k_RR.wav
LargeRoom_ImpResp_48k_LL.wav	LargeRoom_ImpResp_48k_LR.wav
LargeRoom_ImpResp_48k_RL.wav	LargeRoom_ImpResp_48k_RR.wav

Lecture Hall

The lecture hall is located on the New Mexico State University campus in the Klipsch School of Electrical and Computer Engineering (Thomas and Brown, Room 104). Fig. 17 shows the schematic of recording setup with dimensions. The available impulse response files are

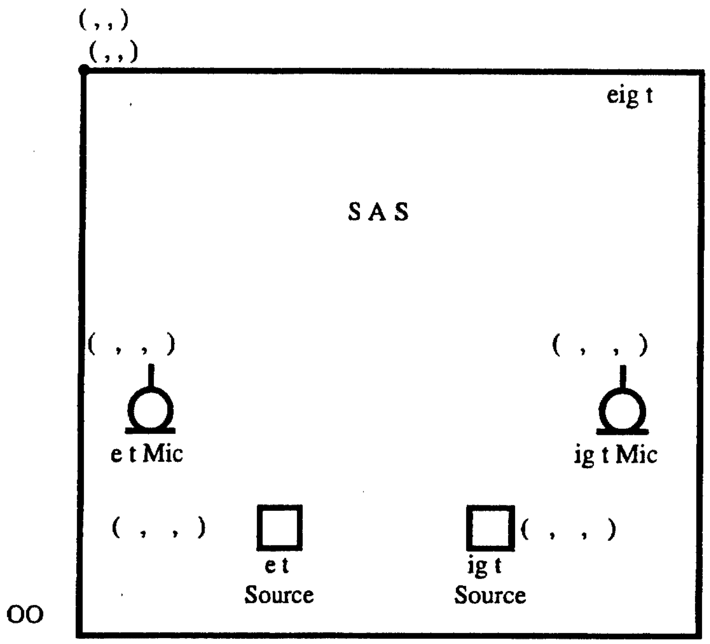


Figure 17: Schematic of Lecture Hall recording setup.

`Lecture_ImpResp_16k_LL.wav`	`Lecture_ImpResp_16k_LR.wav`
`Lecture_ImpResp_16k_RL.wav`	`Lecture_ImpResp_16k_RR.wav`
`Lecture_ImpResp_48k_LL.wav`	`Lecture_ImpResp_48k_LR.wav`
`Lecture_ImpResp_48k_RL.wav`	`Lecture_ImpResp_48k_RR.wav`

Small Room (Office)

The Small Room is located on the New Mexico State University campus in the Klipsch School of Electrical and Computer Engineering (Goddard Hall, Room 171). Fig. 18 shows the schematic of recording setup with dimensions. The available impulse response files are

`SmallRoom_ImpResp_16k_LL.wav`	`SmallRoom_ImpResp_16k_LR.wav`
`SmallRoom_ImpResp_16k_RL.wav`	`SmallRoom_ImpResp_16k_RR.wav`
`SmallRoom_ImpResp_48k_LL.wav`	`SmallRoom_ImpResp_48k_LR.wav`
`SmallRoom_ImpResp_48k_RL.wav`	`SmallRoom_ImpResp_48k_RR.wav`

Stairwell

The stairwell is located on the New Mexico State University campus in the Klipsch School of Electrical and Computer Engineering on the second floor. Fig. 19 shows the schematic of recording setup with dimensions. The available impulse response files are

`Stairwell_ImpResp_16k_LL.wav`	`Stairwell_ImpResp_16k_LR.wav`
`Stairwell_ImpResp_16k_RL.wav`	`Stairwell_ImpResp_16k_RR.wav`
`Stairwell_ImpResp_48k_LL.wav`	`Stairwell_ImpResp_48k_LR.wav`
`Stairwell_ImpResp_48k_RL.wav`	`Stairwell_ImpResp_48k_RR.wav`

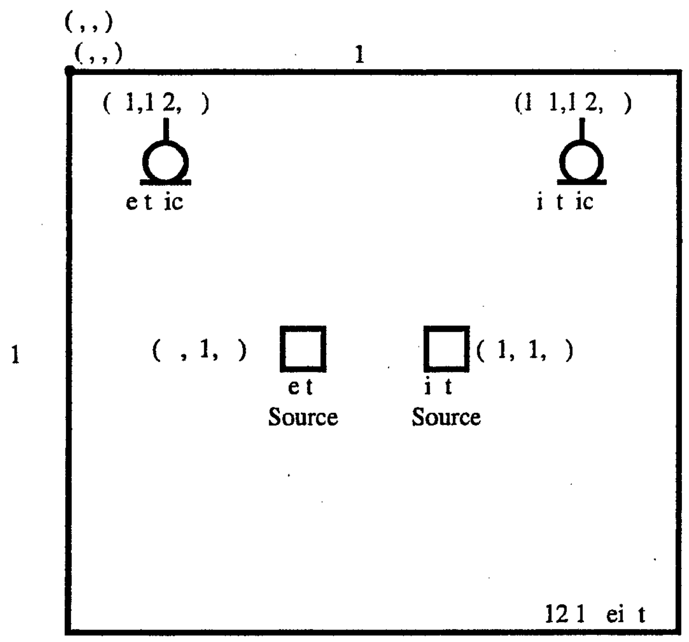


Figure 18: Schematic of Small Room recording setup.

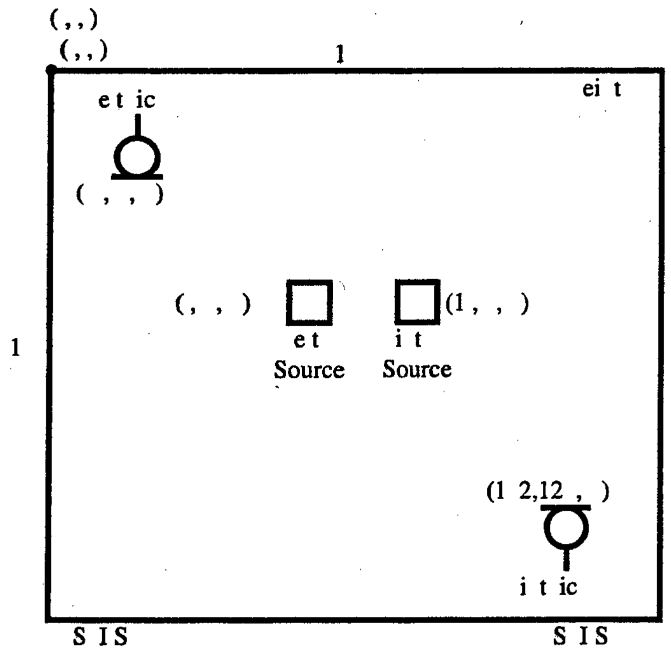


Figure 19: Schematic of Stairwell recording setup.

Study Lounge
The study lounge is located on the New Mexico State University campus in the Klipsch School of Electrical and Computer Engineering (Thomas and Brown, Room 102). Fig. 20 shows the schematic of recording setup with dimensions. The available impulse response files

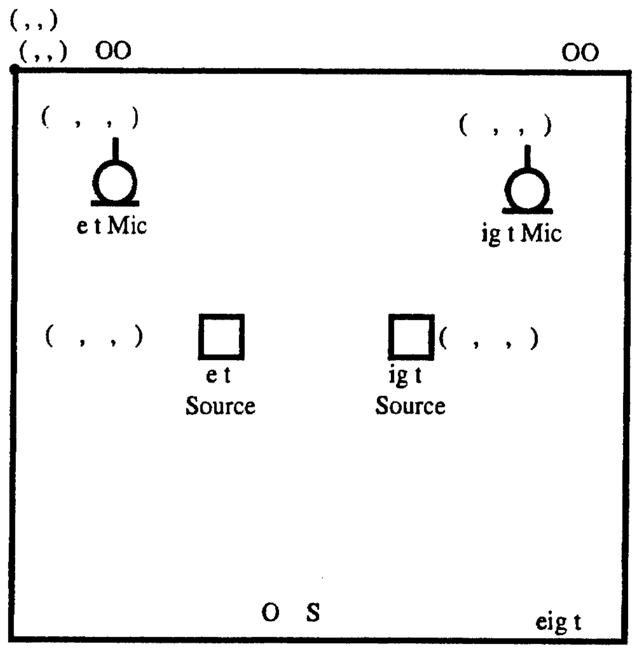


Figure 20: Schematic of Study Lounge recording setup.

are

`Study_ImpResp_16k_LL.wav`	`Study_ImpResp_16k_LR.wav`
`Study_ImpResp_16k_RL.wav`	`Study_ImpResp_16k_RR.wav`
`Study_ImpResp_48k_LL.wav`	`Study_ImpResp_48k_LR.wav`
`Study_ImpResp_48k_RL.wav`	`Study_ImpResp_48k_RR.wav`

4 Results and Discussion

In this section, we report results of our research. These results include

- Optimal parameter selection for the modified MRFD algorithm and resulting SIRI results
- Algorithm performance in terms of various simulated room sizes and reverberation times
- Algorithm performance when mixture signals have additive noise present
- Algorithm performance and processing times for short mixture signal lengths and impact on real-time operation
- Performance of separating (enhancing) one speech signal from a background of additive noise.

4.1 Selection of Optimal Parameters for the MRFD Algorithm

We consider the selection of optimal parameters for the modified MRFD algorithm. The four most important parameters that affect the separation performance are the: block length (or maximum filter length), T; length of the un-mixing filter, W_{ij} at each stage given by the set $\mathbf{Q} = \{Q_1, \ldots, Q_{SG}\}$; adaptive step size, $\tilde{\mu}$; and number of iterations, L. Generally, the longer the un-mixing filter is, ultimately the better the SIRI. However, the permutation problem constrains $Q < T$. Generally, we choose at the last stage, a maximum un-mixing filter length equal to the block length, $Q_{SG} = T$. The parameter set for the multistage method includes the number of stages (SG) and the filter length Q_i at each stage.

Careful selection of the step size is necessary to obtain good performance from gradient-based adaptive algorithms used in decorrelation and speech separation tasks [16]. In this context, the step size controls the convergence rate and the misadjustment, which form a tradeoff pair. Fast convergence, i.e. convergence with smaller L, requires a bigger step size, which may lead to large misadjustment. On the contrary, we can achieve much better SIRI using a smaller step size but at a cost of decreased convergence speed. The number of iterations, L, is one of the main computational bottlenecks in the MRFD algorithm. We ideally want to choose the smallest value L in order to achieve fast separation while maintaining acceptable SIRI. There are also other parameters that play less important roles in the MRFD algorithm. These are

- Initial value of un-mixing filter matrix, $\mathbf{W}^{(0)}(\omega)$ — An adaptive system is stable in the sense that it can be driven to the steady state without regard to the choice of initial status. However, the objective function, $J(\omega)$ has local minima, therefore we set the initial un-mixing matrices $\mathbf{W}^{(0)}(\omega)$ to be identity matrices over frequencies as is typical in the literature.
- Number of iterations, K — Building an over-determined least squares problem requires the number of superblocks, $K \geq 4$ in order to generate enough equations. Note that

larger K can guarantee better separation performance due to better (longer-term) statistical measures of the mixtures as shown in Fig.3. However, K also depends on N, the length of the input mixtures. For a fixed N, K has to be properly chosen to give good average estimates of the PSD matrix, $\hat{\mathbf{R}}_x(\omega, k)$. In most of our experiments, the input signals have a duration of $7 - 14$ sec, accordingly, we choose $4 \leq K \leq 7$ (assuming an 8kHz sample rate).

- Overlap factor, β — The required rate of the properly sampled short-time spectrum representation needs to be $2-4$ times higher than for the equivalent TD representation of the speech signal itself [5]. Thus, we set the overlap factor $\beta = 0.5$ for Hamming window.

- Total number of blocks, U, and number of blocks in each superblock, M — These two parameters are determined by the selection of the other parameters N, β, T, and K with $U = N/(\beta T)$ and $M = U/K$.

4.1.1 Block Length T

An optimal T is desired to satisfy the short-time statistical assumptions regarding the speech signals while maintaining high spectral resolution of the un-mixing filters (in our setting $Q_{SG} = T$). In order to measure the SIRI with varying block length under the same mixing process, we make use of metrics given in (3.26) and (3.27) for real-world recordings because the metrics given in (3.23) and (3.24) require $P = T$, i.e. varying mixing filter length at the same time. In our experiment, we randomly choose 20 pairs of speech signals from the TIMIT speech corpus. Each pair is mixed through the simulated room impulses, shown in Fig. 21, with mixing filter length 2048. We fix other parameters as $\tilde{\mu} = 1.0, L = 100$, and update the multistage parameter set according to $\mathbf{Q} = [T/4,\ T]^T$. Fig. 22 shows the SIRI in dB with T varying over the range 128 to 16384. We find the optimal block length $T = 2048$.

4.1.2 Multistage Parameter Set Q

Here, we fix the block length $T = 2048$, the adaptive step size $\tilde{\mu} = 1.0$, and randomly choose 13 combinations of SG and Q_q. The average SIRI of 20 pairs of speech mixtures with the sources randomly selected from TIMIT is measured for each case and listed in Table. 3. There are two sets of experimental results for each case: the result in the first row of each cell are the average SIRIs of MRFD, while in the second row are average SIRIs of modified MRFD listed in italic font. One can see that the modifications to MRFD result in separation performance that is 1-2dB better than that of the original MRFD at every stage for each case. Furthermore, the optimal performance is obtained in the two-stage case with $Q_1 = 500$, and $Q_2 = 2048$ through modified MRFD algorithm. Again we note these modifications not only improve SIRI but require no additional complexity (modification #2) and reduce cost (modification #1).

Table 3: Average SIRI for different combinations of $[Q_1, \ldots, Q_{SG}]$

Stage 1		Stage 2		Stage 3		Stage 4		Stage 5	
Q_1	SIRI	Q_2	SIRI	Q_3	SIRI	Q_4	SIRI	Q_5	SIRI
	(dB)		(dB)		(dB)		(dB)		(dB)
300	15.83								
	17.31								
200	14.82	500	14.76						
	15.79		*16.21*						
300	15.83	1024	14.66						
	17.31		*16.28*						
500	16.12	1024	14.49						
	17.70		*15.97*						
300	15.83	2048	18.27						
	17.31		*19.49*						
500	16.12	2048	18.41						
	17.70		*19.63*						
700	16.07	2048	18.40						
	17.38		*19.04*						
300	15.84	500	14.79	1024	14.28				
	17.31		*16.50*		*16.01*				
500	16.12	1200	14.41	2048	18.24				
	17.70		*15.72*		*19.44*				
500	16.12	800	14.57	1000	14.16	1500	13.45		
	17.70		*16.09*		*15.61*		*14.87*		
256	15.35	800	14.60	1024	14.22	2048	18.08		
	16.50		*15.88*		*15.48*		*18.90*		
300	15.84	500	14.79	700	14.38	1024	14.18	2048	13.42
	17.31		*16.50*		*16.14*		*15.94*		*14.99*
500	16.12	800	14.57	1000	14.16	1500	13.44	2048	15.99
	17.70		*16.09*		*15.61*		*14.87*		*18.70*

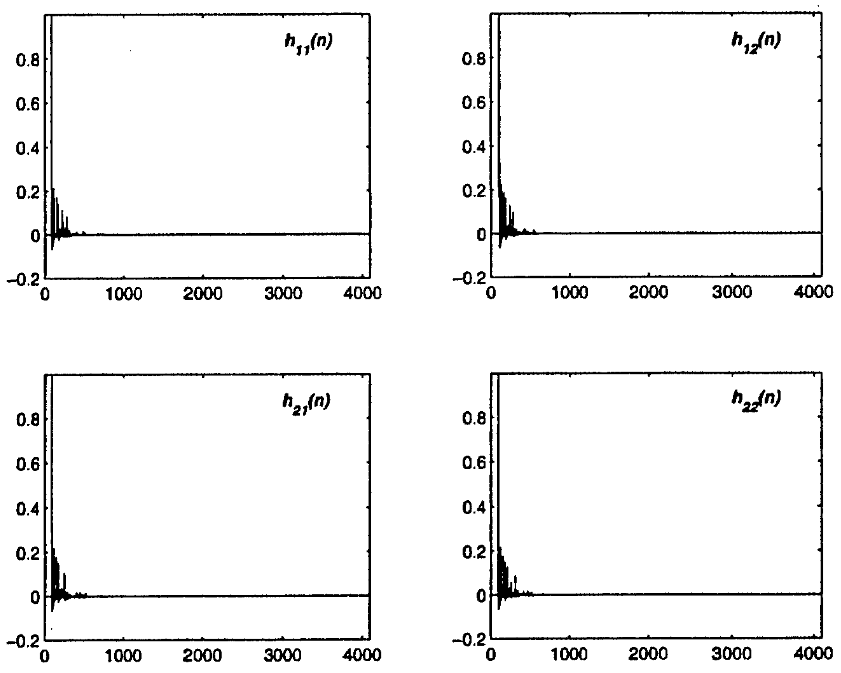


Figure 21: Simulated room impulse responses.

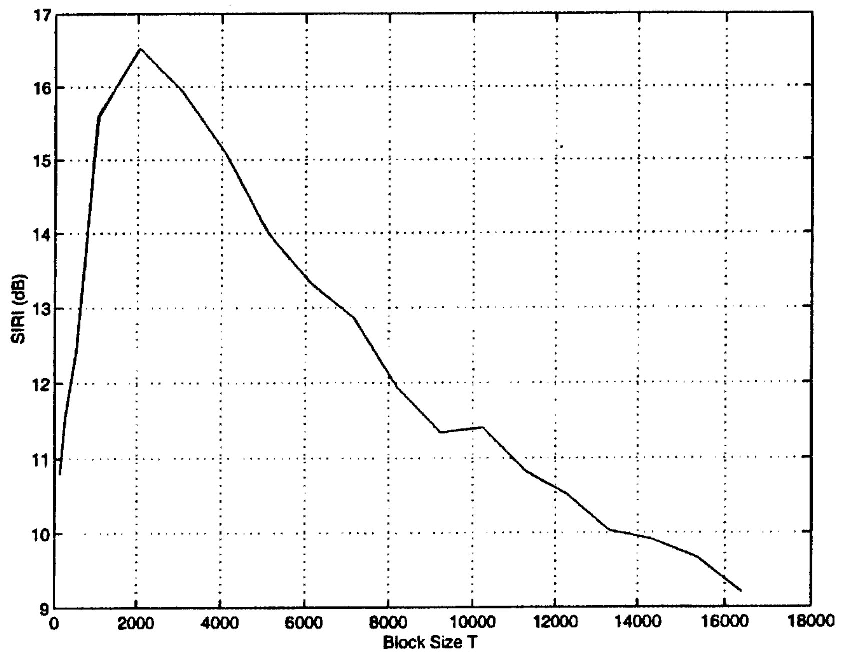


Figure 22: Average SIRI versus block length T.

4.1.3 Adaptive Step Size $\tilde{\mu}$

Fixing $T = 2048$, and $Q = [500\ 2048]$, we examine the SIRI with varied adaptive step size $\tilde{\mu}$ in the range from 10^{-6} to 2.0. The results are shown in Fig. 23. As one can see that the optimal adaptive step size is achieved at $\tilde{\mu} = 1.0$. Also, from Fig. 24, it is easy to see that if $\tilde{\mu}$ is too small, the adaptive algorithm takes too long to converge as shown in the first four plots. On the other hand, if $\tilde{\mu}$ is too large (> 1.0), the algorithm may become unstable. This leads to failure as can be seen in the last two plots of Fig. 24.

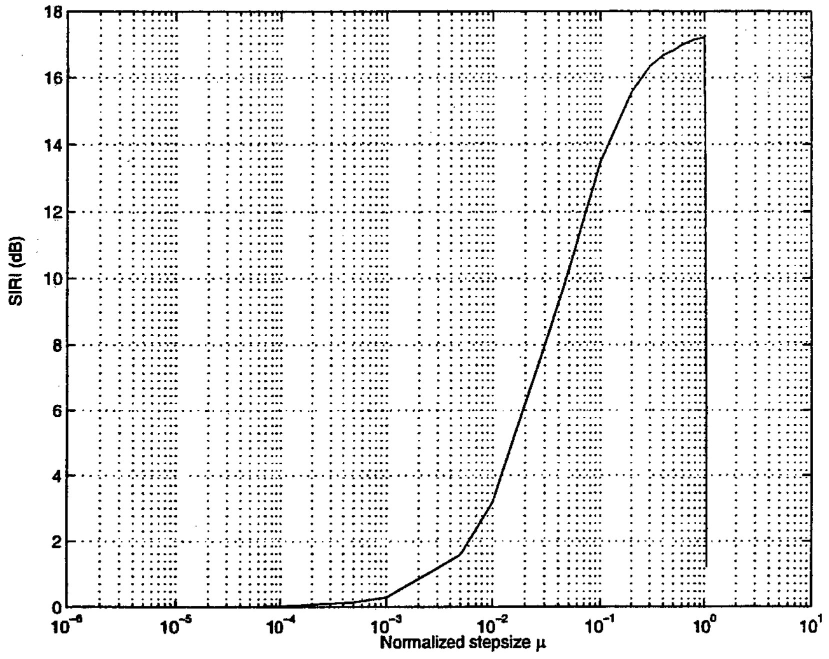


Figure 23: Average SIRI versus adaptive step size $\tilde{\mu}$.

4.1.4 Number of Iterations L

It is well-known that for the MRFD algorithm, more filter update iterations will lead to a higher SIRI. However, a large L requires more computational time, which limits tracking a time-varying environment and real-time applications. Fig. 25.(a) shows the SIRI plot with increasing L and other parameters fixed as $T = 2048$, $Q = [256\ 2048]$, and $\tilde{\mu} = 1.0$. Fig. 25b shows the processing time versus the number of iterations L (Matlab 6.0, Dell Precision Workstation 330 with 1.8GHz P4 and 1GB RAM). In the simulations we are mainly interested in the SIRI performance with acceptable processing time, we therefore choose $L = 100$.

4.1.5 Summary

Optimal parameters for the modified MRFD that yield good separation performance at reasonable computational cost are shown in Table 4. Unless otherwise noted, these parameters are used in all the following experiments.

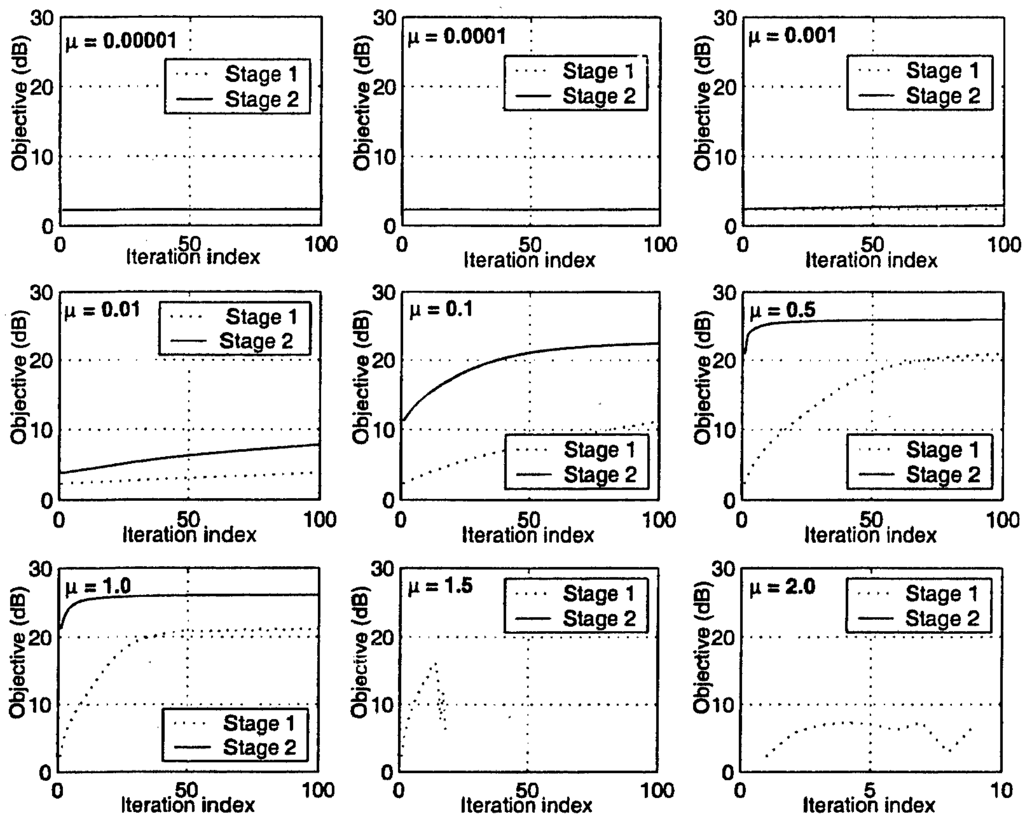


Figure 24: SIRI versus iteration index with different step size $\tilde{\mu}$.

Table 4: Simulation parameters

Parameter	Value
Overlap factor, β	0.5
Number of superblocks, K	6
Number of iterations, L	100
Normalized step size, $\tilde{\mu}$	1.0
Multistage parameter set, $\mathbf{Q}$	$[500,\ 2048]^T$
Block size, T	2048
Initial un-mixing filter matrix for all stages, $\mathbf{W}^{(0)}(\omega)$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

4.2 Results on the Effect of Room Size and Reverberation Time

We simulate the room impulse responses $h_{ji}(n)$ for various reflection coefficients, ρ which determines the "liveliness" of the acoustic environment. For simplicity, we set ρ the same for all six walls in the simulated room. The average SIRI using the modified MRFD of 20 random pairs of speech mixtures are measured and pictured in Fig. 26. As expected, higher

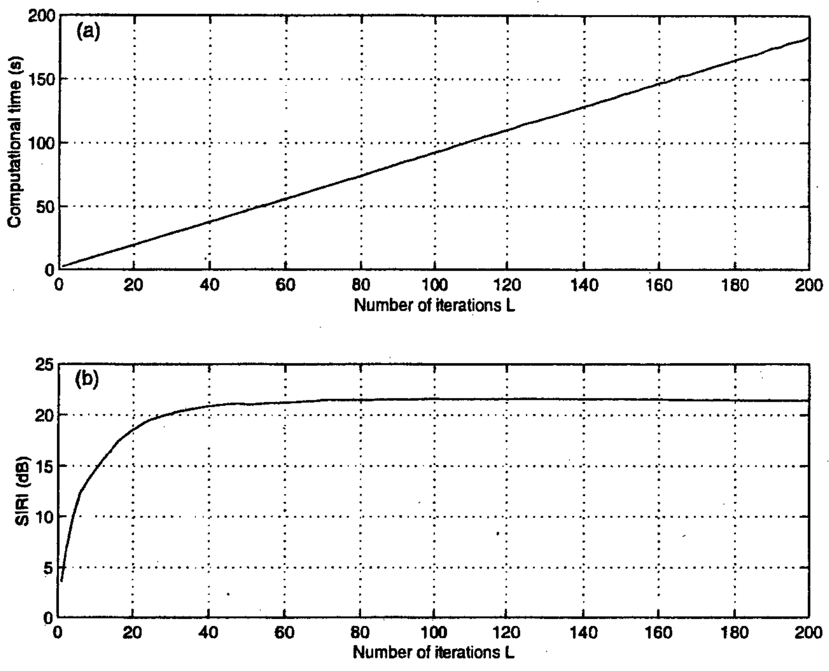


Figure 25: The tradeoff between separation performance and processing time.

SIRIs are achieved for the room with smaller reflection coefficient, i.e. less reverberation.

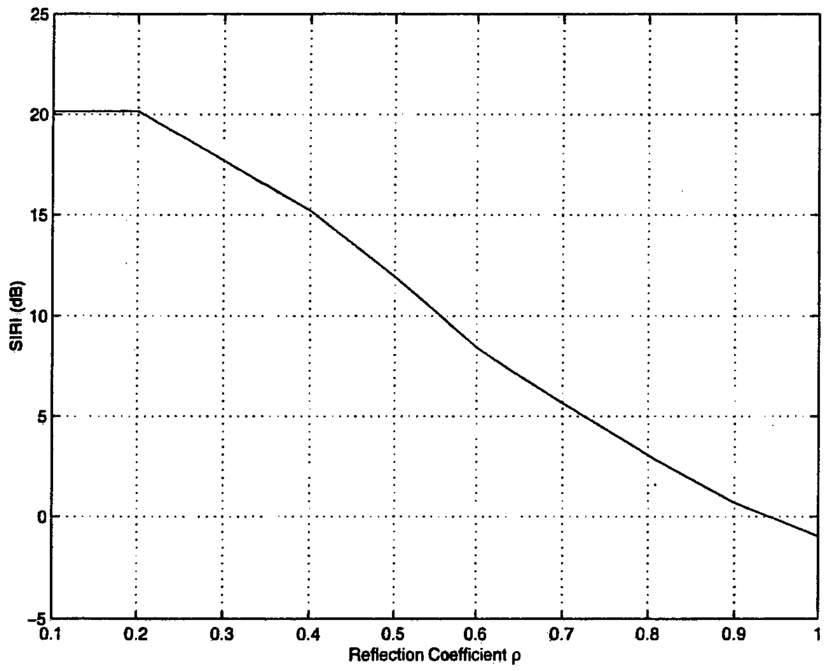


Figure 26: SIRI performance in simulated room with different reverberation.

4.2.1 Various Room Sizes

In this section, the room geometry in Fig. 9 varies with a in the range of 4m to 30m according to Table 5. The reflection coefficient is $\rho = 0.3$ for each of six walls. Fig. 27 shows the resulting SIRI obtained by varying room size parameter a. As one can see, the separation performance improves with increasing room size, which is also due to the decreased reverberation as in the case of varying reflection coefficients.

Table 5: Simulated room parameters.

Parameters	Values
Source1 Coordinates	$(\frac{a}{2} - 1, a - 2, 1.5)$
Source2 Coordinates	$(\frac{a}{2} + 1, a - 2, 1.5)$
Microphone 1 coordinates	$(\frac{a}{2} - 1, a, 1.7)$
Microphone 2 coordinates	$(\frac{a}{2} + 1, a, 1.7)$
Room Dimensions	$(a, a, \frac{a}{2})$

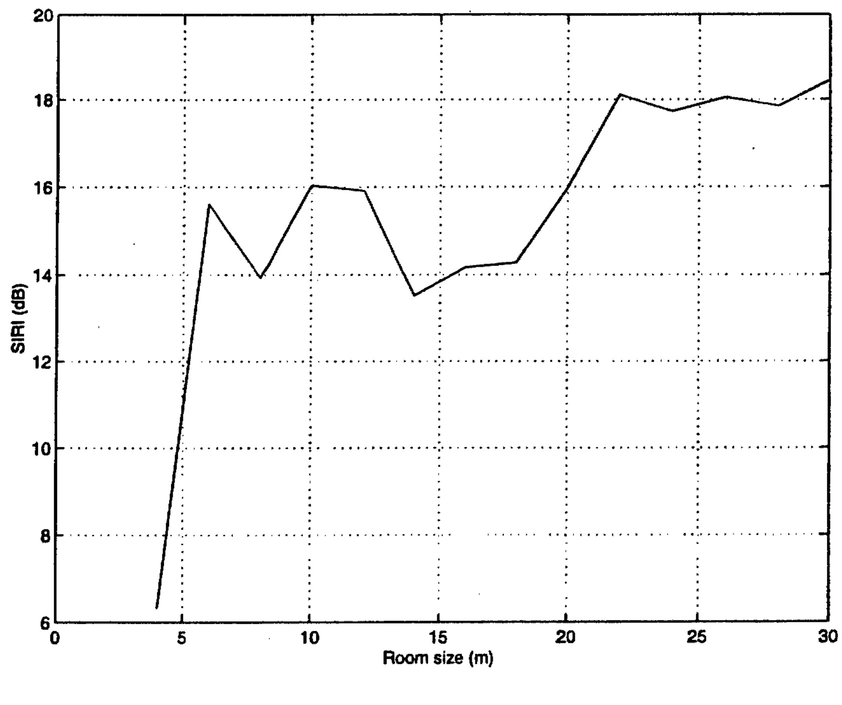


Figure 27: SIRI performance in simulated room with varying room size.

4.3 Results with Two Speakers and Additive Background Noise

If a noise source is present in addition to the two speech sources, separation of the speech signals will still proceed but this noise will be present in each separated signal. In addition, SIRI gracefully degrades as the SNR decreases (noise power increases). Fig. 28 illustrates the experimental setup and Fig. 29 shows the SIRI performance as a function of SNR. This finding and motivates the requirement for additional denoising of correlated noise sources from the speech signals in order to obtain accurate ASR.

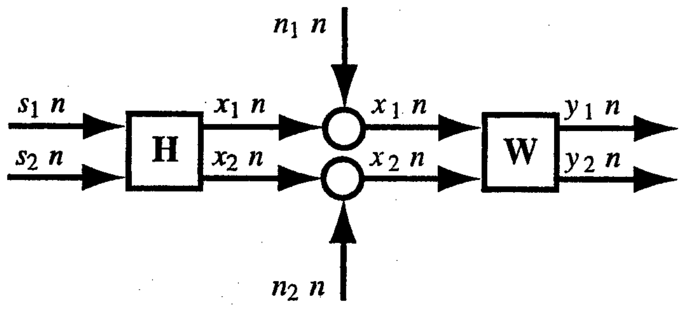


Figure 28: Block diagram of BSS with background AWGN.

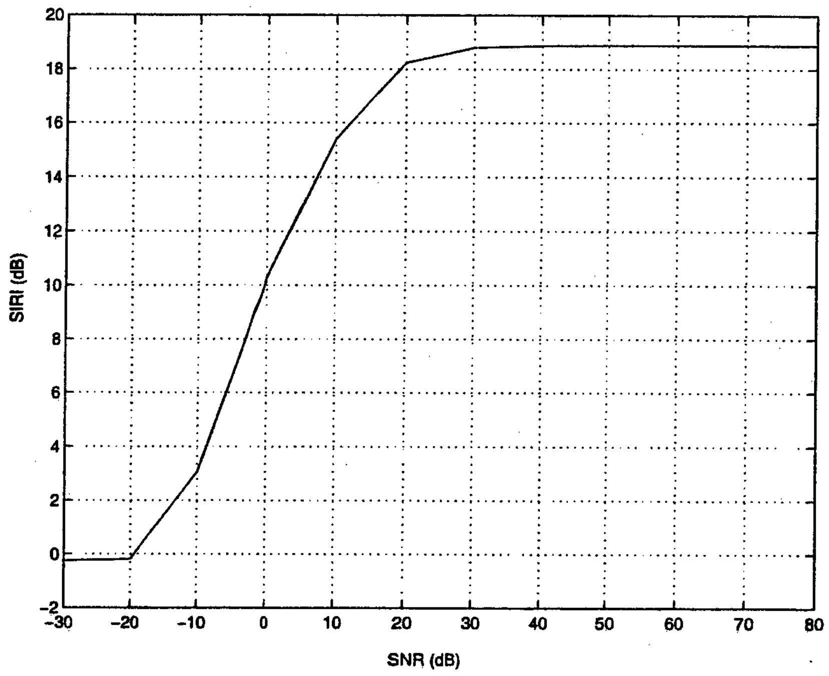


Figure 29: SIRI performance in various levels of background noise.

From Fig. 29, we can see that 3 – 10dB SIRI can be achieved when the mixtures are buried in background noise (−10dB < SNR < 0dB) as in a vehicle or aircraft or other noisy environment. When the average power ratio of the mixed speech signals to the additive noise exceeds 10dB, the MMRFD algorithm yields a 15.5dB SIRI in an average sense. We can therefore conclude that MMRFD is robust under background, wideband noise conditions.

4.4 Results with Short Mixture Signals

Up to now, all simulations have used mixing filters with fixed impulse responses. However, the channel impulse responses may change dramatically even for the speaker just turning his/her head 10 – 20 degrees, or leaning backward a couple of inches [17]. In practical environments, the algorithm must be able to rapidly adapt in these dynamic situations.

We assume that during the time that multiple, second order statistics are estimated, the mixing filters remain approximately the same [10]. The question we ask, however, is can we collect enough samples for adaption before the change of mixing filters even under a slowly varying acoustical environment? Or, equivalently what is the amount of data needed by the MRFD algorithm for successful signal separation? In order to answer these questions, we conduct an experiment to test the potential of an online application of the MRFD algorithm. Fig. 30 shows the SIRI for various input signal lengths with varying number of iterations L. As expected, we see that with increasing signal length, the performance of MRFD increases. Quantitatively, the performance of MRFD approaches about 14dB SIRI with just 2sec of input at sampling rate of 16kHz. We note that this is the upper bound of SIRI for most other off-line algorithms with input signal lengths of 30 to 50 seconds [18], [19], [20].

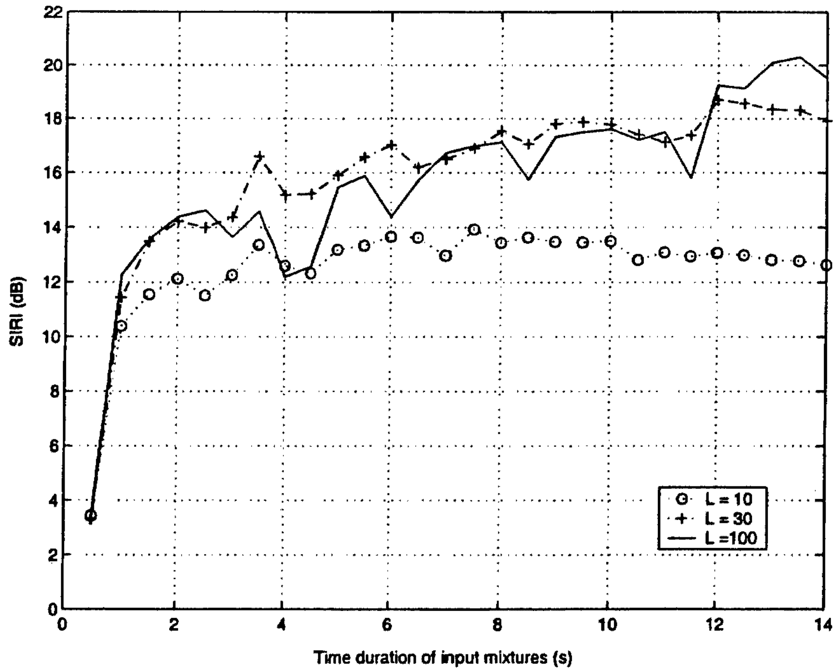


Figure 30: SIRI versus length of input mixtures.

Even if there is enough data available, another question is posed in [17]: Is it possible to perform the necessary computation in real time? Our computational benchmarks indicate 30sec of input speech mixtures recorded in a real room can be successfully separated in 75 sec with more than 20dB enhancement (Matlab 6.0, Dell Precision 330 with 1.8GHz P4, 1GB RAM). Fig. 31 shows the processing time versus the length of input for three different numbers of adaptive iterations. We can see from Fig. 30 and Fig. 31 that properly reducing the amount of iterations can dramatically save processing time while maintaining relatively high SIRI. In addition, we have indications of near-real-time performance when MRFD is implemented in *C* (as previously described in Section 3.7) and run on a PC. Therefore real-time performance on a dedicated DSP should not be a problem.

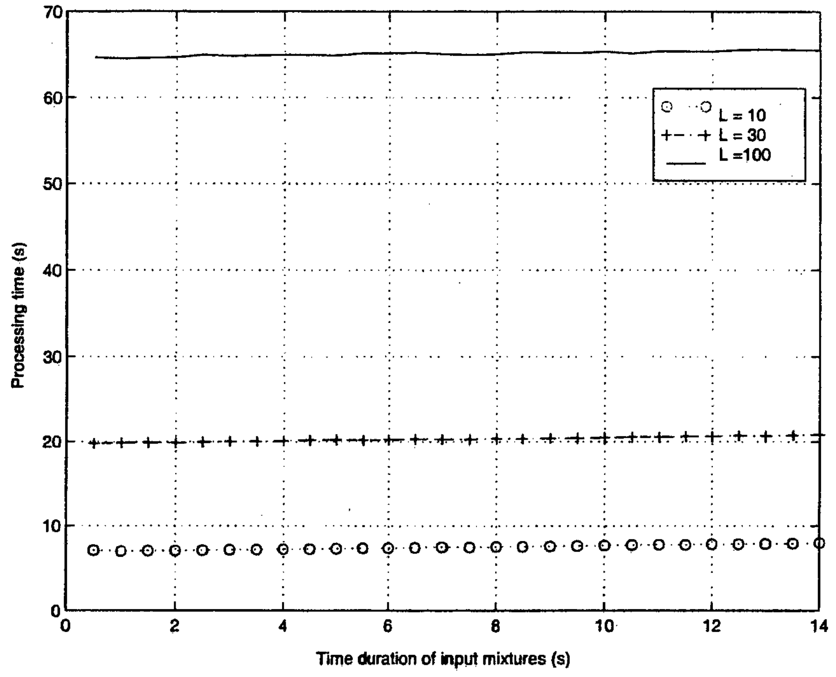


Figure 31: Processing time versus length of input mixtures.

4.5 Results with One Speaker in the Presence of Background Noise

If a background noise (white noise or even music) is present in a single speaker's speech signal, MRFD can be used to separate the single speech signal from the background noise with SIRIs comparable to the two speaker case. This results from the basic assumption that the sources are uncorrelated. The system model is illustrated in Fig. 32, where $s_2(n)$ is now a White Gaussian Noise (WGN) signal.

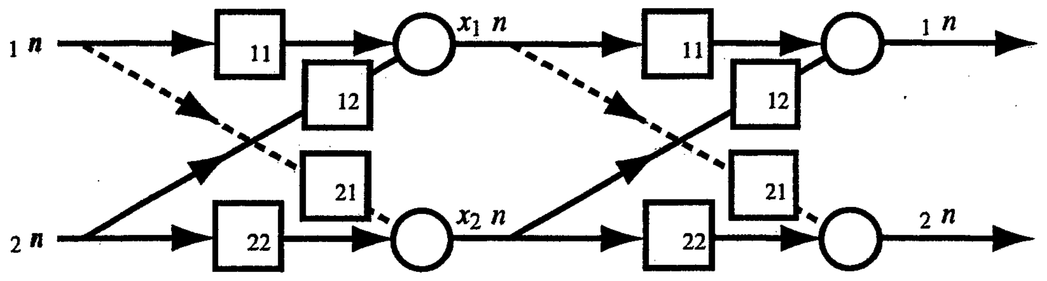


Figure 32: Block diagram of noise-cancellation problem.

Simulations results indicate that we achieve 16.4dB measured by (3.23) and (3.24) when the speech signal power is approximately the same as the noise noise power, i.e. input SNR = 0dB. In the case of knowing that there is no cross-channel contribution from $s_1(n)$ to $x_2(n)$, i.e. pure noise, we can apply a constraint of $W_{21} = 0$, which increase the output SNR by 5.03dB to 21.46dB. Fig. 33 is a plot of output SNR versus the SNR of noise-corrupted input speech signals $\mathbf{x}(n)$. From this performance plot, we can see when the speech signal is buried in the background noise (-20dB < SNRi < 0dB), we can achieve $23 - 30$dB SNR improvement (SNRI), which shows successful noise-cancellation. When the the source speech signal dominates the mixture (SNRi > 40dB), no SNRI can be seen since there is little noise to be cancelled in the first place.

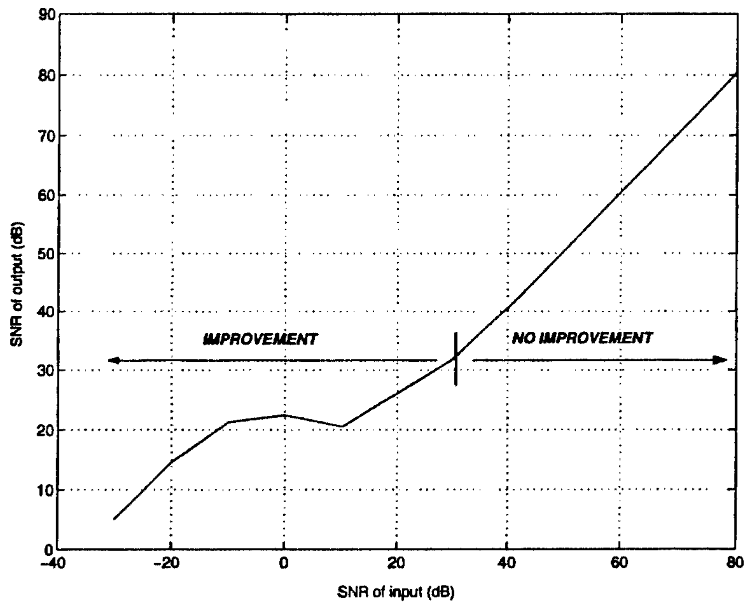


Figure 33: Signal-to-Interference Ratio Improvement as a function of noise after two-channel BSS processing.

5 Conclusions

In the blind source/speech separation problem, we attempt to separate out individual, speech signals from a background of other undersired speech signals and other acoustic noise sources. In this research, we have gained an understanding of the source separation problem for convolutional mixtures and developed expertise with the Multiresolution Frequency-Domain algorithm. We have improved the MRFD algorithm by developing methods for exploiting symmetry in order to reduce computation by 50% and developing initialization strategies for better performance. We have experimentally determined optimal parameters for the algorithm and studied the effects of room size and liveliness on separation performance. Simulations with both digitally-filtered and mixed speech signals as well as real-world signals indicate good performance with this algorithm with 10-20dB improvements to the Signal-to-Interference Ratio of these sources. In the important case of speech enhancement, i.e. blind suppression of background noise from a desired speech signals, we have been able to increase SIRI upwards of 20dB. Finally, our work has demonstrated a robustness to separating speech signals in the presence of noise. Furthermore, we have carefully examined the potential for real-time processing with this algorithm and have concluded that good performance is possible but with at least a 2sec. delay due to the inherent "block processing" nature of the algorithm.

In addition, as part of the deliverables for this grant, we have developed a blind speech signal corpus or database containing speech signal mixtures in a variety of acoustical environments. This database is now available on the Internet at

`http://www.ece.nmsu.edu/~pdeleon/BSS/index.html`

for free download.

6 Recommendations

We have demonstrated that speech signals can be significantly enhanced despite little knowledge regarding the speech signal itself, background noise signals, or acoustical environment. The primary constraints are the need for multi-channel input signals and a 2sec. delay in producing the enhanced signal. Given these constraints it may be possible to apply this work to improving automatic speech recognition in adverse environments, although it is unknown at this time, the direct impact SIRI has on ASR accuracy.

In most environments, there will be more than two sources of speech and noise. In this case, we require extension of the two-channel MRFD to the multi-channel case involves generalization of (3.14). While this is fairly straightforward, two problems must be investigated in order for multi-channel BSS algorithms to be realized to their full potential. First, the multi-channel BSS algorithms will be very computationally complex with complexity growing as the square of the number of channels. With more than two acoustic sources, it is not expected that the algorithms could be computed in real-time with current and near-term DSP technology. This problem might be addressed with a specialized *analog* co-processor. Second, the permutation inconsistency, where separated sources may contain spectral energy at certain frequencies from the other source and vice versa, will be exacerbated in the multi-channel case. In this case, there are now many, many more permutations which result from simple row swaps of the unmixing filter matrix. This problem may be resolved through the multi-resolution approach (or some version of it). The idea is that with short unmixing filter lengths, Q, the solution is smooth in the frequency domain and for the most part the permutations are self-aligning. In order to increase spectral resolution and subsequently SIRI, Q is increased. Thus for the multi-channel BSS algorithm, we may require more stages to bring the filter length, Q, up slowly in order to maintain permutation consistency.

7 References

References

[1] D. Morgan, E. George, L. Lee, and S. Kay, "Cochannel speaker separation by harmonic enhancement and suppression," *IEEE Trans. Speech Audio Proc.*, vol. 5, pp. 407–424, Sep. 1997.

[2] K. Yen and Y. Zhao, "Adaptive co-channel speech separation and recognition," *IEEE Trans. Speech Audio Proc.*, vol. 7, pp. 138–151, Mar. 1999.

[3] L. Parra and C. Spence, "Convolutive blind separation of non-stationary sources," *IEEE Trans. Speech Audio Proc.*, vol. 8, pp. 320–327, May 2000.

[4] M. Ikram and D. Morgan, "A multiresolutional approach to blind separation of speech signals in a reverberant environment," *Proc. Int. Conf. Acoust., Speech, and Sig. Proc. (ICASSP)*, 2001.

[5] L. Rabiner and R. Schafer, *Digital Processing of Speech Signals.* Prentice-Hall, Inc., 1978.

[6] M. Ikram and D. Morgan, "Exploring permutation inconsistency in blind separation of speech signals in a reverberant environment," *Proc. Int. Conf. Acoust., Speech, and Sig. Proc. (ICASSP)*, 2000.

[7] L. Parra and C. Spence, "Separation of non-stationary natural signals." in *Independent Components Analysis: Principles and Practice*, eds. Richard Everson and Stephen Roberts, Cambridge University Press, 2001.

[8] S. Amari, A. Cichocki, and H. Yang, "A new learning algorithm for blind signal separation." in *Advances in Neural Information Processing Systems*, eds. David S. Touretzky, Michael C. Mozer, and Michael E. Hasselmo, MIT Press, 1996.

[9] T. Lee, A. Bell, and R. Orglmeister, "Blind source separation of real world signals," *Proc. Int. Conf. Neural Networks*, 1997.

[10] L. Parra and C. Spence, "Online convolutive source separation of non-stationary signals," *J. VLSI Sig. Proc.*, vol. 26, August 2000.

[11] F. Ehlers and H. Schuster, "Blind separation of convolutive mixtures and an applications in automatic speech recognition in a noisy environment," *IEEE Trans. Sig. Proc.*, vol. 45, pp. 2608–2612, Oct. 1997.

[12] D. Schobben, K. Torkkola, and P. Smaragdis, "Evaluation of blind signal separation methods," *Proc. ICA and BSS*, 1999.

[13] J. Allen and D. Berkley, "Image method for efficiently simulating small-room acoustics," *J. Acoust. Soc. Am.*, vol. 65, no. 4, pp. 943–950, 1979.

[14] W. Fisher, G. Doddington, and K. Goudie-Marshall, "The DARPA speech recognition research database: specifications and status," *Proc. DARPA Workshop on Speech Recog.*, pp. 93–99, 1986.

[15] S. Orfanidis, *Introduction to Signal Processing.* Upper Saddle River, NJ: Prentice Hall, 1996.

[16] B. Krongold and D. Jones, "Blind source separation of nonstationary convolutively mixed signals," *Proc. 10th IEEE Workshop on Stat. Sig. and Array Proc.*, 2000.

[17] S. Haykin, ed., *Unsupervised Adaptive Filtering: Volume 1 Blind Source Separation.* New York, NY: John Wiley and Sons, Inc., 2000.

[18] K. Torkkola, "Blind separation of convolved sources based on information maximization," *IEEE Workshop Neural Networks for Sig. Proc.*, 1996.

[19] S. Amari, S. Douglas, A. Cichocki, and H. Yang, "Multichannel blind deconvolution and equalization using the natural gradient," *Proc. 1st IEEEWorkshop on Sig. Proc. Advances in Wireless Comm.*, 1997.

[20] R. Lambert and A. Bell, "Blind separation of multiple speakers in a multipath environment," *Proc. Int. Conf. Acoust., Speech, and Sig. Proc. (ICASSP)*, 1997.

Appendix A - Derivation of the Cost Function Gradient

We know that for any real-valued function $f(z)$ of a complex variable z, the gradient $\nabla_z f(z)$ can be obtained by

$$\begin{aligned}\nabla_z f(z) &= \tfrac{\partial f(z)}{\partial \Re(z)} + \tfrac{\partial f(z)}{\partial \Im(z)} \\ &= 2\tfrac{\partial f(z)}{\partial \bar{z}}.\end{aligned} \tag{A.1}$$

We can expand (3.9) as

$$\begin{aligned}\hat{\mathbf{R}}_{\mathbf{y}}(\omega,k) &= \begin{bmatrix} W_{11}(\omega) & W_{12}(\omega) \\ W_{21}(\omega) & W_{22}(\omega)\end{bmatrix}\begin{bmatrix}\hat{R}_{x_{11}}(\omega,k) & \hat{R}_{x_{12}}(\omega,k) \\ \hat{R}_{x_{21}}(\omega,k) & \hat{R}_{x_{22}}(\omega,k)\end{bmatrix}\begin{bmatrix} W_{11}^*(\omega) & W_{21}^*(\omega) \\ W_{12}^*(\omega) & W_{22}^*(\omega)\end{bmatrix} \\ &= \begin{bmatrix} \times & \hat{R}_{y_{12}}(\omega,k) \\ \hat{R}_{y_{21}}(\omega,k) & \times \end{bmatrix}\end{aligned} \tag{A.2}$$

which is a sequence of Hermitian matrices with

$$\begin{aligned}\hat{R}_{y_{12}}(\omega,k) &= \left[W_{11}(\omega)\hat{R}_{x_{11}}(\omega,k) + W_{12}(\omega)\hat{R}_{x_{21}}(\omega,k)\right] W_{21}^*(\omega) \\ &\quad + \left[W_{11}(\omega)\hat{R}_{x_{12}}(\omega,k) + W_{12}(\omega)\hat{R}_{x_{22}}(\omega,k)\right] W_{22}^*(\omega) \\ \hat{R}_{y_{21}}(\omega,k) &= \left[W_{21}(\omega)\hat{R}_{x_{11}}(\omega,k) + W_{22}(\omega)\hat{R}_{x_{21}}(\omega,k)\right] W_{11}^*(\omega) \\ &\quad + \left[W_{21}(\omega)\hat{R}_{x_{12}}(\omega,k) + W_{22}(\omega)\hat{R}_{x_{22}}(\omega,k)\right] W_{12}^*(\omega).\end{aligned} \tag{A.3}$$

Therefore,

$$\begin{aligned}\nabla_{\mathbf{W}} J(\omega) &= 2\frac{\partial J(\omega)}{\partial \mathbf{W}^*(\omega)} \\ &= 2\sum_{k=1}^{K}\left[\frac{\partial|\hat{R}_{y_{12}}(\omega,k)|^2}{\partial \mathbf{W}^*(\omega)} + \frac{\partial|\hat{R}_{y_{21}}(\omega,k)|^2}{\partial \mathbf{W}^*(\omega)}\right] \\ &= 2\sum_{k=1}^{K}\left[\hat{R}_{y_{12}}^*(\omega,k)\frac{\partial \hat{R}_{y_{12}}(\omega,k)}{\partial \mathbf{W}^*(\omega)} + \hat{R}_{y_{21}}^*(\omega,k)\frac{\partial \hat{R}_{y_{21}}(\omega,k)}{\partial \mathbf{W}^*(\omega)}\right] \\ &= 2\sum_{k=1}^{K}\left\{\hat{R}_{y_{21}}(\omega,k)\mathbf{A} + \hat{R}_{y_{12}}(\omega,k)\mathbf{B}\right\} \\ &= 2\sum_{k=1}^{K}\mathbf{V}(\omega,k)\mathbf{D} \\ &= 2\sum_{k=1}^{K}\mathbf{V}(\omega,k)\mathbf{W}(\omega)\hat{\mathbf{R}}_{\mathbf{x}}(\omega,k).\end{aligned} \tag{A.4}$$

with

$$\mathbf{A} =$$

$$\begin{bmatrix} 0 & 0 \\ W_{11}(\omega)\hat{R}_{x_{11}}(\omega,k) + W_{12}(\omega)\hat{R}_{x_{21}}(\omega,k) & W_{11}(\omega)\hat{R}_{x_{12}}(\omega,k) + W_{12}(\omega)\hat{R}_{x_{22}}(\omega,k)\end{bmatrix},$$

$$\mathbf{B} =$$

$$\begin{bmatrix} W_{21}(\omega)\hat{R}_{x_{11}}(\omega,k) + W_{22}(\omega)\hat{R}_{x_{21}}(\omega,k) & W_{21}(\omega)\hat{R}_{x_{12}}(\omega,k) + W_{22}(\omega)\hat{R}_{x_{22}}(\omega,k) \\ 0 & 0 \end{bmatrix},$$

and

$$\mathbf{D} =$$

$$\begin{bmatrix} W_{11}(\omega)\hat{R}_{x_{11}}(\omega,k) + W_{12}(\omega)\hat{R}_{x_{21}}(\omega,k) & W_{11}(\omega)\hat{R}_{x_{12}}(\omega,k) + W_{12}(\omega)\hat{R}_{x_{22}}(\omega,k) \\ W_{21}(\omega)\hat{R}_{x_{11}}(\omega,k) + W_{22}(\omega)\hat{R}_{x_{21}}(\omega,k) & W_{21}(\omega)\hat{R}_{x_{12}}(\omega,k) + W_{22}(\omega)\hat{R}_{x_{22}}(\omega,k) \end{bmatrix}.$$

Appendix B - Personnel Associated with Research Effort

Principal Investigator
Dr. Phillip L. De Leon
Associate Professor
New Mexico State University
Klipsch School of Electrical and Computer Engineering
Las Cruces, New Mexico 88003-8001
(505) 646-3771
`pdeleon@nmsu.edu`

Technical Staff
Dr. Krist Petersen
College Associate Professor and Assistant Department Head
New Mexico State University
Klipsch School of Electrical and Computer Engineering
Las Cruces, New Mexico 88003-8001
(505) 646-4932
`kpeterse@nmsu.edu`

Mr. Jay Chen
Research Assistant
New Mexico State University
Klipsch School of Electrical and Computer Engineering
Las Cruces, New Mexico 88003-8001
(505) 646-1911
`xichen@ece.arizona.edu`

(Mr. Chen in now at the University of Arizona pursuing is M.S.E.E.)

Mr. Ning Chen
Research Assistant
New Mexico State University
Klipsch School of Electrical and Computer Engineering
Las Cruces, New Mexico 88003-8001
(505) 646-1911
`nchen@nmsu.edu`

(Mr. Chen was awarded a Master of Science in Electrical Engineering (M.S.E.E.) from NMSU for his thesis entitled, "Analysis of the Multiresolution Frequency Domain Algorithm for Blind Source Separation." He is now pursuing his Ph.D. at the Georgia Institute of Technology.)

Mr. Steve Spence
Research Assistant
New Mexico State University
Klipsch School of Electrical and Computer Engineering
Las Cruces, New Mexico 88003-8001
(505) 646-1911
`sspence@nmsu.edu`

(Mr. Spence currently at New Mexico State University pursuing is M.S.E.E. His thesis investigates real-time implementation issues of the modified MRFD algorithm.)

Appendix C - Publications Resulting from Research Effort

The following publications resulted from this research effort. Many of the results of these theses have been included in this report.

N. Chen, "Analysis of the Multiresolution Frequency Domain Algorithm for Blind Source Separation," *Master's Thesis, New Mexico State University*, 2001.

S. Spence, "Real-Time Implementation of a Modified Multiresolution, Frequency-Domain, Blind Speech Separation Algorithm," *Master's Thesis, New Mexico State University*, 2002.

Appendix D - MATLAB Simulation Codes

mrfd208.m

```
%-----------------------------------------------------------------------------
% CO-CHANNEL CONVOLUTIVE MIXED SPEECH SEPARATOR
%
% This MATLAB code implements the blind speech separation algorithm of Ikram
% and Morgan [1].  The problem is as follows.  Given two signals, x1 and
% x2 each containing a convolutive mixture of two source speech signals, s1 and
% s2 produce separated speech signals, y1 and y2 such that y1 ~ s1, y2 ~ s2.
% This is a blind separation problem since we only have x1 and x2 and no
% other additional information.  The approach assumes the uncorrelatedness
% between the original source signals.  We then attempt to diagonalize the
% estimated Power Spectral Density (PSD) matrices at multiple time segments
% to decorrelate the mixtures in frequency-domain (avoiding the deconvolution
% in time-domain) to achieve the separation.
%-----------------------------------------------------------------------------
% Notes
% 1. This MATLAB implements the modified version of the algorithm in [1] as
%    given in Table. 3 in this thesis.
%
% 2. All signals stored as column vectors.
% -----------------------------------------------------------------------------
% Version History
%
% 2.05 (05 Sep. 01) Cleaned up the mess and added comments.
%
% 2.04 (26 Aug. 01) Selected the optimal parameters, and changed K to 6.
%
% 2.03 (25 Aug. 01) Fixed a bug on the implementation of the symmetric
%                   constraint, the correct one is W(omega)=W(Q-omega+1)
%                   for omega=1:Q, so we need to seek W(omega) for
%                   omega=1:T/2+1. Added the objective measurement to evaluate
%                   how well the diagonalization worked at each iteration.
%
% 2.02 (14 Aug. 01) Modified the MRFD by the reinitialization of W(omega) at
%                   each stage.
%
% 2.01 (27 Jul. 01) Changed the required input signal length of 51 seconds as
%                   the experiments in [1] to any length N. Fixed K=4, thus
%                   calculate U=N/(beta*T) and M=U/K.
%
% 2.00 (27 Jun. 01) Major rewrite with Prof. De Leon.
%      (1) Changed the truncation of the un-mixing filters from frequency-domain
%          to time-domain to smooth the frequency responses and solve the
%          permutation problem.
%      (2) Added the symmetric constraint on the un-mixing filters in frequency-
%          domain so that we only needed to seek W(omega) for omega=1:T/2, and
%          reduce the computational complexity by half.
%      (3) Replaced most of the loop commands by matrix-based calculation to
%          reduce the computational time.
%      (4) Tried to normalize the Gradient by its own power instead of the
%          average power of mixtures, but the SIRI performance was not as good
%          as the original normalization approach.
%
% 1.03 (1 Apr. 01) Modified the approach to synthesize the output signals from
%          overlap-save to overlap-adding, by which the separated signals sound
%          normal though no separation yet.
%
% 1.02 (10 Mar. 01) Fixed a bug on the Hamming window function. Measured the
%          separation of instantaneous mixtures.
%
% 1.01 (5 Feb. 01) Added the intermediate separation process before the second
%      stage, and also update the PSD matrices at multiple time segments.
%
% 1.00 (Jan 30, 01) First version based on the original understanding or MRFD
%      reported by Ikram and Morgan in [1]. In this version:
%      (1). We took the gradient term as given in [2] without derivation.
%      (2). We used all the parameters given in [1], and Q = [500, 2048]'. In
%           the first stage, Q1=500, we only seek the un-mixing matrices for
%           the first Q1 frequencies, while for the rest (Q2-Q1) frequency bins,
%           we set W(omega)=[1, 0; 0, 1].
%      (3). For the synthesis of the separated signals, we first passed the
%           mixture vectors through the un-mixing filter matrices W(omega) in
%           frequency-domain (multiplication), then converted the output signals
%           back to the time-domain, finally we divided them by the window
%           function and took use of the overlap-save approach.
%
% [1] M. Ikram and D. Morgan, "A multiresolution Approach to Blind Separation
%     of Speech Signals in A Reverberant Environment", In Proc. IEEE ICASSP,
%     Pages 2757-2760, 2001.
% [2] L. Parra and C. Spence, "Convolutive Blind Separation of Non-Stationary
%     Sources", IEEE Transactions on Speech and Audio Processing, 8(3):320-327,
%     May 2000.
%
%-----------------------------------------------------------------------------
% Bug Reports
% Report all bugs to Prof. Phillip De Leon (pdeleon@nmsu.edu)
```

```
%-----------------------------------------------------------------------------

clear; close all; clc
%-----------------------------------------------------------------------------
% Set initial parameters
T = 2048; % Basic block length
beta = 0.5; % Overlapping factor
K = 6; % The total number of superblocks
alfa = 1.0; % Normalized step size
L = 100;  % Number of iterations
Q = [500; 2048]; % Multistage parameters

%-----------------------------------------------------------------------------
% Real-world mixtures
mix1 = 'c:\nchen\research\mrfd\altin1.wav'; mix2 =
'c:\nchen\research\mrfd\altin2.wav';
% The directory used to save the separated signals
out = 'c:\nchen\research\mrfd\out\final_mrfd\altin\';
% Read mixtures into vectors
[xt1,Fs]=wavread(mix1); [xt2,Fs]=wavread(mix2);

%-----------------------------------------------------------------------------
% Determine the parameters: N, U, and M
N = min(length(xt1), length(xt2)); % Signal length in samples
N = floor(N/T)*T;
xt1 = xt1(1:N); xt2 = xt2(1:N);     % Truncate signal to int. number of blocks
U = floor(N/(beta*T)-1);            % Number of blocks
M = floor(U/K);                     % Number of blocks in each superblock

%-----------------------------------------------------------------------------
% Initialize Hamming window
t = [0:T-1]'; w = 0.54 - 0.46*cos(2*pi*t/(T-1));

%-----------------------------------------------------------------------------
% Short-Time Fourier Transform
xw1 = zeros(T,1); xw2 = zeros(T,1);

x1 = zeros(T,N/(beta*T)); x2 =zeros(T,N/(beta*T));

for m = 1:U
   block = beta*T*(m-1);
   xw1 = w.*xt1(block+1:block+T);
   xw2 = w.*xt2(block+1:block+T);% The column vector of windowed mixtures
   x1(:,m) = fft(xw1); % x1, x2 are (T * MK) matrices with column vector...
   x2(:,m) = fft(xw2); % as the FFT of each block
```

```
end;

%-----------------------------------------------------------------------------
% Initializing the un-mixing filter matrices
W = zeros(2,2,T); W(1,1,:) = 1; W(1,2,:) = 0; W(2,1,:) = 0;
W(2,2,:) = 1;
% Four un-mixing filter vectors
W11 = zeros(T,1); W12 = W11; W21 = W11; W22 = W11; Y1 =
zeros(T,1); Y2 = zeros(T,1);

%----------------------------------------------------------------------------%
%----------------------------------------------------------------------------%
%%                         MULTISTAGE ITERATION                             %%
for q = 1:length(Q)
    % Compute the covariance
    Rx=zeros(2,2,T/2+1,K); % only calculate Rx on omega=1:T/2+1 due to symmetry
    for k = 1:K
       block = M*(k-1);
       Rx(1,1,:,k) = sum(x1(1:T/2+1,block+[1:M]).* ...
                     conj(x1(1:T/2+1,block+[1:M])),2)./M;
       Rx(1,2,:,k) = sum(x1(1:T/2+1,block+[1:M]).* ...
                     conj(x2(1:T/2+1,block+[1:M])),2)./M;
       Rx(2,1,:,k) = conj(Rx(1,2,:,k)); %Hermitian property of the PSD matrices
       Rx(2,2,:,k) = sum(x2(1:T/2+1,block+[1:M]).* ...
                     conj(x2(1:T/2+1,block+[1:M])),2)./M;
    end;

    % Average power of the mixtures at each frequency
    Mu = zeros(T/2+1,1);
    for omega = 1:T/2+1
       Fronorm = 0;
       for k=1:K
         Fronorm = Fronorm + norm(Rx(:,:,omega,k),'fro').^2; %Frobenius-norm
       end;
       Mu(omega) = alfa/Fronorm; % Step size
    end;

    % Reinitializing the un-mixing filter matrices at each stage
    Ws = zeros(2,2,T);
    Ws(1,1,:) = 1; Ws(1,2,:) = 0; Ws(2,1,:) = 0; Ws(2,2,:) = 1;

    for l = 1:L % Iteration for L times
       for omega = 1:T/2+1 % Seek W(omega) at every frequency bin
          Gradt = zeros(2,2);% Initializing Gradient
          for k = 1:K
```

```
                V = Ws(:,:,omega)*Rx(:,:,omega,k)*Ws(:,:,omega)';
                V = V - diag(diag(V)); %cost function
                Gradt = Gradt + V*Ws(:,:,omega)*Rx(:,:,omega,k);
             end;
             Gradt = 2 * (Gradt-diag(diag(Gradt))); % Gradient
             Ws(:,:,omega) = Ws(:,:,omega) - Mu(omega)*Gradt; % Update
          end;
          %% Apply the symmetric constraint
          Ws(:,:,T/2+2:T) = conj(Ws(:,:,T/2:-1:2));
          %% Truncation of W(omega) in time-domain
          W11=reshape(Ws(1,1,:),T,1); W12=reshape(Ws(1,2,:),T,1);
          W21=reshape(Ws(2,1,:),T,1); W22=reshape(Ws(2,2,:),T,1);
          W11=ifft(W11); W12=ifft(W12); W21=ifft(W21); W22=ifft(W22); % to TD
          W11=fft(W11(1:Q(q)),T); W12=fft(W12(1:Q(q)),T);  % to frequency domain
          W21=fft(W21(1:Q(q)),T); W22=fft(W22(1:Q(q)),T);
          Ws(1,1,:)=W11; Ws(1,2,:)=W12; Ws(2,1,:)=W21; Ws(2,2,:)=W22;
       end; % End of L iterations

       % Current Stage Separation
       for m = 1:U
           Y1 = W11.*x1(:,m) + W12.*x2(:,m);
           Y2 = W21.*x1(:,m) + W22.*x2(:,m);
           x1(:,m) = Y1;
           x2(:,m) = Y2;
       end;
       % Update the overall un-mixing filter matrices
       for omega = 1:T
          W(:,:,omega) = Ws(:,:,omega) * W(:,:,omega);
       end;

end;
%%                      END of MULTISTAGE ITERATION                           %%
%-------------------------------------------------------------------------------%
%-------------------------------------------------------------------------------%

%-------------------------------------------------------------------------------
% Inverse Fourier Transform and synthesize separated signals by Overlap-adding
block = beta*T; yy1 = zeros(T,1); yy2 = yy1; y1 = yy1; y2 = yy1;
for m = 1:U
   yy1 = real(ifft(x1(:,m)));
   yy2 = real(ifft(x2(:,m)));
   L1 = length(y1);
   L2 = length(y2);
   y1 = [y1(1:L1-block);y1(L1-block+1:L1)+yy1(1:block);yy1(block+1:T)];
   y2 = [y2(1:L1-block);y2(L1-block+1:L1)+yy2(1:block);yy2(block+1:T)];
```

```
end; e = 10^(-7);
y1 = y1./(max(abs(y1))+e); y2 = y2./(max(abs(y2))+e); % Normalize

%--------------------------------------------------------------------------
% Save separated signals
wavwrite(y1,Fs,strcat(out,'altin_T',num2str(T),'_1.wav'));
wavwrite(y2,Fs,strcat(out,'altin_T',num2str(T),'_2.wav'));
```

mrfdSIM203.m

```
%--------------------------------------------------------------------------
% CO-CHANNEL CONVOLUTIVE MIXING & UN-MIXING (MRFD) SIMULATOR
%
% This MATLAB code simulates the convolutive mixing process using the simulated
% impulse responses generated by the "image method" [1] and implements the
% blind speech separation algorithm of MRFD by Ikram and Morgan [2].  The
% problem is as follows.  Given two source signals, s1 and s2, and four mixing
% filters simulated by the "image method", we generate the mixtures x1 and x2
% by x_i = s_i*h_ij + s_j*h_ij, where "*" is the linear convolution operator.
% Then we implement MRFD based on the statistics collected from the x1 and x2
% and produce separated speech signals, y1 and y2 such that y1 ~ s1, y2 ~ s2.
% Finally, we determine the Signal-to-Noise-Ratio Improvement (SIRI).
%
% [1]  J. B. Allen and D. A. Berkley, "Image Method for efficiently simulating
%      small-room acoustics", J. Acoust. Soc. Am., 65(4):943--950, 1979.
%
% [2]  M. Ikram and D. Morgan, "A multiresolution Approach to Blind Separation
%      of Speech Signals in A Reverberant Environment", In Proc. IEEE ICASSP,
%      Pages 2757-2760, 2001.
%--------------------------------------------------------------------------
% Notes
% 1. This MATLAB implements the modified version of the algorithm in [1] as
%    given in Table. 3 in this thesis, and measures the average SIRI
%    performance for the selection of optimal parameters and analysis of
%    MRFD under various reverberant environment.
%
% 2. All signals stored as column vectors.
%
% -------------------------------------------------------------------------
% Version History
%
% 2.03 (Oct. 10, 01) Cleaned up mess and added comments.
%
% 2.02 (Sep. 10, 01) Added the second measurement of SIRI, which could be used
%                        when the mixing filters were unknown.
%
% 2.01 (Sep. 01, 01) Rewrote the main code to be able to measure the average
%                    SIRI of num_sour pairs of mixtures with the source
%                    speeches randomly selected from TIMIT.
%
% 2.00 (Aug. 06, 01) Changed the mixing filters by the simulated impulse
%                    responses of real room generated through the image method
%                    in [3]
%
% 1.02 (Jul. 22, 01) Changed the mixing filters by the impulse responses given
%                    at: Prof. De Leon's BSS website:
%                        http://www.ece.nmsu.edu/~pdeleon/BSS/index.html.
%                    Truncated those impulse responses to the first T samples.
%
% 1.01 (Feb. 28, 01) Added the calculation of Signal to Interference Ratios
%     Improvement(SIRI), the impulse responses were offered by Prof. De Leon,
%     with direct-channel response hii from the speaker i to the telephone
%     transmitter i, and cross-channel response hij (i~=j) from speaker j
%     to telephone i over the phone line, we truncated the impulse responses
%     to get the first T samples.
%
% 1.00 (Jan. 30, 01) First version based on the code mrfd_real.m version 1.0.
%-----------------------------------------------------------------------
% Required routines:  At least one of
%    siri_rho.m (determine SIRI with varying reflection coefficients),
%    siri_rms.m (determine SIRI with varying room size),
%    imag_filt.m (simulate the room impulse response by image method),
%    sroom.m (simulate the room impulse response by image method),
%    lthimage.m (simulate the room impulse response by image method),
%    siri_meas1.m (measure SIRI with known sources and mixing filters),
%    siri_meas2.m (measure SIRI without known sources and mixing filters),
%-----------------------------------------------------------------------

clear; close all; clc;
% ----------------------------------------------------------------------
% Initializing parameters
num_sour = 20; % Number of random pairs for average SIRI measurement
rho = 0.1:0.1:1.0; % Analysis of MRFD for various reflection coefficients
% rho = 0.3 % Fixing rho when analyzing other parameters
T = 2048;
%a = [4:2:30]'; % Analysis of varying room size [a, a, a/2]

% ----------------------------------------------------------------------
% Set work directory
```

```
database = 'c:\nchen\TIMIT'; % Location of TIMIT database
func_path = 'c:\nchen\research\mrfd\reflect';

out_fig = 'C:\nchen\research\mrfd\reflect\ana_reflect.fig';
path = path(path,func_path); % Add function path into Matlab search path

% -----------------------------------------------------------------------
% Get TIMIT database directory structure
eval(strcat('cd(''', database, ''')'));
inpath = dir;  % Get the "dir" structure
n=size(inpath);
inpath = inpath(3:n(1)); %cut the first two elements, which are "." and ".."

% -----------------------------------------------------------------------
% Determine average SIRI
SIRI = zeros(length(rho),1);
for i = 1:num_sour % Determine the SIRI for each pair of mixture
  samps = ceil(600.*rand(1,2));
  path = inpath(samps); % Randomly choose 2 speech signals
  for j = 1:length(rho) %
    SIR = siri_rho(path(1).name, path(2).name, rho(j));
%   RL = [a(j); a(j); a(j)/2]; % Room dimension
%   SIR = siri_rms(path(1).name, path(2).name, RL); % SIRI performance with
    SIRI(j) = SIRI(j) + SIR;                        % varying room size
  end;
end;
SIRI = SIRI./num_sour; % Average SIRI

% -------------------------------------------------------------------
% Plot SIRI versus varying reflection coefficients
figure; plot(rho,SIRI); xlabel('Reflection Coefficient \rho');
ylabel('SIRI (dB)'); grid; saveas(gcf,out_fig);
```

siri_rho.m

```
function SIRI = siri_rho(source1, source2, rho);
%-----------------------------------------------------------------------
% SIRI PERFORMANCE ANALYSER (For varying reflection coefficients)
%
% This MATLAB code implements the MRFD under various reverberant environment
% by varying reflection coefficients in the "image method" simulation.
%
% Call Syntax: SIRI = siri_rho(source1, source2, rho)
%
```

```
% INPUT arguments:
% Name: source1/2
% Type: string
% Description: File name of source speech signal (.wav)
%
% Name: rho
% Type: scalar
% Description: Reflection Coefficient
%
% OUTPUT arguments:
% Name: SIRI
% Type: scalar
% Description: Signal-to-Interference Ratio Improvement
%
% Creation Date: Aug. 06, 2001
% Last Revision: Sept. 10, 2001
%-------------------------------------------------------------------

%-------------------------------------------------------------------
% Set initial parameters
T = 2048; % Basic block length
beta = 0.5; % Overlapping factor
K = 6; % The total number of superblocks
alfa = 1.0; % Normalized step size
L = 100;  % Number of iterations
Q = [500; 2048]; % Multistage parameters

%-------------------------------------------------------------------
% Simultate Mixing filters
[h11,h12,h21,h22] = imag_filt(rho); H11 = fft(h11); H12 =
fft(h12); H21 = fft(h21); H22 = fft(h22);

%-------------------------------------------------------------------
% Source Signals
[st1,f1] = wavread(source1);  [st2,f2] = wavread(source2);

%-------------------------------------------------------------------
% Determine the parameters: N, U, and M
N = min(length(xt1), length(xt2)); % Signal length in samples
N = floor(N/T)*T;
st1 = st1(1:N); st2 = st2(1:N);     % Truncate signal to int. number of blocks
U = floor(N/(beta*T)-1);            % Number of blocks
M = floor(U/K);                     % Number of blocks in each superblock

%-------------------------------------------------------------------
```

```
% Initialize Hamming window
t = [0:T-1]'; w = 0.54 - 0.46*cos(2*pi*t/(T-1));

%------------------------------------------------------------------------------
% Short-Time Fourier Transform and Mixing Process by Circular Convolution
sw1= zeros(T,1);       sw2= zeros(T,1);

s1 = zeros(T,U); s2 = zeros(T,U);

x1 = zeros(T,U); x2 = zeros(T,U); pack;

for m = 1:U
   block = beta*T*(m-1);
   sw1 = w.*st1(block+1:block+T);
   sw2 = w.*st2(block+1:block+T);% The column vector of windowed source signals
   s1(:,m) = fft(sw1);    % s1, s2 are (T * MK) matrics with column vector...
   s2(:,m) = fft(sw2);    % as the FFT of each block
   x1(:,m) = H11.*s1(:,m) + H12.*s2(:,m); % Mixing process by circular
   x2(:,m) = H21.*s1(:,m) + H22.*s2(:,m); % convolution
end;

% %------------------------------------------------------------------------------
% % Mixing Process by Linear Convolution and Short-Time Fourier Transform
% xt1 = conv(st1,h11) + conv(st2,h12);
% xt2 = conv(st1,h21) + conv(st2,h22); % Mixing process by linear convolution
% xt1 = xt1(1:N);    xt2 = xt2(1:N);
%
% % Short-Time Fourier Transform
% sw1 = zeros(T,1); sw2 = zeros(T,1);
% xw1 = zeros(T,1); xw2 = zeros(T,1);
% s1 = zeros(T,num_blk); s2 = zeros(T,num_blk);
% x1 = zeros(T,num_blk); x2 = zeros(T,num_blk);
% pack;
% for m = 1:num_blk
%    block = beta*T*(m-1);
%
%    sw1 = w.*st1(block+1:block+T); % The column vector of windowed source
%    sw2 = w.*st2(block+1:block+T); % signals
%    s1(:,m) = fft(sw1);  % s1, s2 are (T * MK) matrics with column vector...
%    s2(:,m) = fft(sw2);  % as the FFT of each block
%
%    xw1 = w.*xt1(block+1:block+T); % The column vector of windowed mixed
%    xw2 = w.*xt2(block+1:block+T); % signals
%    x1(:,m) = fft(xw1);  % x1, x2 are (T * MK) matrics with column vector...
%    x2(:,m) = fft(xw2);  % as the FFT of each block
% end;

%-----------------------------------------------------------------------
% Initializing the un-mixing filter matrices
W = zeros(2,2,T); W(1,1,:) = 1; W(1,2,:) = 0; W(2,1,:) = 0;
W(2,2,:) = 1;
% Four un-mixing filter vectors
W11 = zeros(T,1); W12 = W11; W21 = W11; W22 = W11;

Y1 = zeros(T,1); Y2 = zeros(T,1);

%----------------------------------------------------------------------%
%----------------------------------------------------------------------%
%%                        MULTISTAGE ITERATION                        %%
for q = 1:length(Q)
    % Compute the covariance
    Rx=zeros(2,2,T/2+1,K); % only calculate Rx on omega=1:T/2+1 due to symmetry
    for k = 1:K
       block = M*(k-1);
       Rx(1,1,:,k) = sum(x1(1:T/2+1,block+[1:M]).* ...
                     conj(x1(1:T/2+1,block+[1:M])),2)./M;
       Rx(1,2,:,k) = sum(x1(1:T/2+1,block+[1:M]).* ...
                     conj(x2(1:T/2+1,block+[1:M])),2)./M;
       Rx(2,1,:,k) = conj(Rx(1,2,:,k)); %Hermitian property of the PSD matrices
       Rx(2,2,:,k) = sum(x2(1:T/2+1,block+[1:M]).* ...
                     conj(x2(1:T/2+1,block+[1:M])),2)./M;
    end;

    % Average power of the mixtures at each frequency
    Mu = zeros(T/2+1,1);
    for omega = 1:T/2+1
       Fronorm = 0;
       for k=1:K
         Fronorm = Fronorm + norm(Rx(:,:,omega,k),'fro').^2; %Frobenius-norm
       end;
       Mu(omega) = alfa/Fronorm; % Step size
    end;

    % Reinitializing the un-mixing filter matrices at each stage
    Ws = zeros(2,2,T);
    Ws(1,1,:) = 1; Ws(1,2,:) = 0; Ws(2,1,:) = 0; Ws(2,2,:) = 1;

    for l = 1:L % Iteration for L times
       for omega = 1:T/2+1 % Seek W(omega) at every frequency bin
          Gradt = zeros(2,2);% Initializing Gradient
```

```
            for k = 1:K
               V = Ws(:,:,omega)*Rx(:,:,omega,k)*Ws(:,:,omega)';
               V = V - diag(diag(V)); %cost function
               Gradt = Gradt + V*Ws(:,:,omega)*Rx(:,:,omega,k);
            end;
            Gradt = 2 * (Gradt-diag(diag(Gradt))); % Gradient
            Ws(:,:,omega) = Ws(:,:,omega) - Mu(omega)*Gradt; % Update
         end;
         %% Apply the symmetric constraint
         Ws(:,:,T/2+2:T) = conj(Ws(:,:,T/2:-1:2));
         %% Truncation of W(omega) in time-domain
         W11=reshape(Ws(1,1,:),T,1); W12=reshape(Ws(1,2,:),T,1);
         W21=reshape(Ws(2,1,:),T,1); W22=reshape(Ws(2,2,:),T,1);
         W11=ifft(W11); W12=ifft(W12); W21=ifft(W21); W22=ifft(W22); % to TD
         W11=fft(W11(1:Q(q)),T); W12=fft(W12(1:Q(q)),T);  % to frequency domain
         W21=fft(W21(1:Q(q)),T); W22=fft(W22(1:Q(q)),T);
         Ws(1,1,:)=W11; Ws(1,2,:)=W12; Ws(2,1,:)=W21; Ws(2,2,:)=W22;
      end; % End of L iterations

      % Current Stage Separation
      for m = 1:U
          Y1 = W11.*x1(:,m) + W12.*x2(:,m);
          Y2 = W21.*x1(:,m) + W22.*x2(:,m);
          x1(:,m) = Y1;
          x2(:,m) = Y2;
      end;
      % Update the overall un-mixing filter matrices
      for omega = 1:T
         W(:,:,omega) = Ws(:,:,omega) * W(:,:,omega);
      end;

end;
%%                      END of MULTISTAGE ITERATION                        %%
%-----------------------------------------------------------------------%
%-----------------------------------------------------------------------%

% The mixing filter
H = zeros(2,2,T); H(1,1,:) = H11;  H(1,2,:) = H12; H(2,1,:) = H21;
H(2,2,:) = H22;

% %---------------------------------------------------------------------
% % Determine SIRI by the first measurement of SIRI with known sources and
% % mixing filters
% SIRI = siri_meas1(T,U,s1,s2,H,W);
```

```
%------------------------------------------------------------------------
% Determine SIRI by the second measurement of SIRI without known sources and
% mixing filters
%------------------------------------------------------------------------
% The contribution when one source is "on", the other is "off"
xt11 = conv(st1,h11);  xt21 = conv(st1,h21); % Contribution of source 1
xt12 = conv(st2,h12);  xt22 = conv(st2,h22); % Contribution of source 2
xt11 = xt11(1:N);  xt21 = xt21(1:N); % Contribution of source 1
xt12 = xt12(1:N);  xt22 = xt22(1:N); % Contribution of source 2

%------------------------------------------------------------------------
% Short-Time Fourier Transform
xw11 = zeros(T,1); xw12 = zeros(T,1); %
xw21 = zeros(T,1); xw22 = zeros(T,1); %
x11 = zeros(T,num_blk); x12 = zeros(T,num_blk); %
x21 = zeros(T,num_blk); x22 = zeros(T,num_blk); %
pack; %
for m = 1:num_blk
   block = beta*T*(m-1);

   xw11 = w .* xt11(block+1:block+T);%
   xw12 = w .* xt12(block+1:block+T);%
   xw21 = w .* xt21(block+1:block+T);%
   xw22 = w .* xt22(block+1:block+T);% The column vectors of windowed mixtures

   x11(:,m) = fft(xw11); %
   x12(:,m) = fft(xw12); %
   x21(:,m) = fft(xw21); % x11, x12, x21, x22 are (T * MK) matrics with column
   x22(:,m) = fft(xw22); % vector as the FFT of each block
end;

%------------------------------------------------------------------------
% Convert x_ij into one matrix
x(:,:,1) = x11; x(:,:,2) = x12; x(:,:,3) = x21; x(:,:,4) = x22;

SIRI = siri_meas2(T,U,x,W);
```

imag_filt.m

```
function [h11,h12,h21,h22] = imag_filt(rho)
%------------------------------------------------------------------------
% ROOM IMPULSE RESPONSE SIMULATOR
%
% This MATLAB code simulates the acoustic impulse responses in a small room.
```

```
%
% Call Syntax: [h11,h12,h21,h22] = imag_filt(rho)
%
% INPUT arguments:
% Name: rho
% Type: scalar
% Description: Reflection coefficient
%
% OUTPUT arguments:
% Name: h_ij
% Type: vector
% Description: Impulse response from source j to microphone i
%
% AUTHORS: Ning Chen, Dr. De Leon
% Version: 1.0
% Creation Date: Jul. 31, 2001
% Last Revision:
% References:
%----------------------------------------------------------------------------

%----------------------------------------------------------------------------
% Set initial parameters
NPTS = 2048; T = 1/16000; C = 344;
unit = 0.3048; % feet -> meter convert unit

%----------------------------------------------------------------------------
% Vector radius to receiver in smaple preiods
Rr1 = [2.23  ;  2.74;  1.68]./(T*C); %
Rr2 = [2.83  ;  2.74;  1.68]./(T*C);

% Vector radius to source in smaple preiods
Rs1 = [2.13  ;   0.91;   1.52]./(T*C); %
Rs2 = [3.35  ;   0.91;   1.52]./(T*C);

% Vector of box dimensions in smaple periods
RL = [5.06; 3.41; 2.44]./(T*C);

%----------------------------------------------------------------------------
% Vector of six wall reflection coefs
BETA =  [rho, rho, rho; rho, rho, rho];

%----------------------------------------------------------------------------
% % Case 2: when rho is fixed to 0.3 and room size is varying
% x = RL(1);
% % Vector radius to receiver in smaple preiods
% Rr1 = [x/2-1; x/2+1; 1.7]./(T*C);
% Rr2 = [x/2+1; x/2+1; 1.7]./(T*C);
%
% % Vector radius to source in smaple preiods
% Rs1 = [x/2-1; x/2-1; 1.5]./(T*C);
% Rs2 = [x/2+1; x/2-1; 1.5]./(T*C);
%
% % Vector of box dimensions in smaple periods
% RL = RL./(T*C); % RL as a parameter with unit of 'm'
%
% % Vector of six wall reflection coefs
% BETA =  [.3,.3,.3; .3,.3,.3];

%------------------------------------------------------------------
h11 = sroom(Rs1,Rr1,RL,BETA,NPTS); %
h12 = sroom(Rs2,Rr1,RL,BETA,NPTS); %
h21 = sroom(Rs1,Rr2,RL,BETA,NPTS); %
h22 = sroom(Rs2,Rr2,RL,BETA,NPTS);

%------------------------------------------------------------------
% Normalization
h11 = h11./max(abs(h11)); h12 = h12./max(abs(h12)); %
h21 = h21./max(abs(h21)); h22 = h22./max(abs(h22));

%------------------------------------------------------------------
% %Plot of mixing-filter impulse responses
%t = 1:NPTS;
%subplot(2,2,1); plot(t,h11,'r');
%subplot(2,2,2); plot(t,h12,'g');
%subplot(2,2,3); plot(t,h21,'b');
%subplot(2,2,4); plot(t,h22,'k');
```

sroom.m

```
function HT = sroom(Rr,Rs,RL,BETA,NPTS)
%**************************************************************************
% Subroutine to calculate a room impulse response.
%
% Input Arguments:
%   Name: Rr
%   Type: 3*1 real vector
%   Description: Vector radius to receiver in smaple preiods = length/(C*T)
%
%   Name: Rs
%   Type: 3*1 real vector
%   Description: Vector radius to source in smaple preiods = length/(C*T)
%
%   Name: RL
%   Type: 3*1 real vector
%   Description: Vector of box dimensions in smaple preiods = length/(C*T)
%
%   Name: BETA
%   Type: 6*1 real vector
%   Description: Vector of six wall reflection coefs (0 < BETA < 1)
%
%   Name: NPTS
%   Type: integer number
%   Description: Number of points of HT to be computed
%
% Output Arguments:
%   Name: HT
%   Type: vector
%   Description: Impulse Response
%
% AUTHORS: Ning Chen, Dr. De Leon
% Version: 1.0
% Creation Date: Jul. 31, 2001
% Last Revision:
% References:
%*****************************************************************

% Set initial parameters
HT = zeros(NPTS,1); NR = zeros(3,1);
DELP = zeros(8,1);%Vector of eight source to image distances in sample periods
PERM = [-1 -1 -1 -1  1  1  1  1;...
        -1 -1  1  1 -1 -1  1  1;...
        -1  1 -1  1 -1  1 -1  1];

PERM2 = sign(PERM+1);

dis = norm(Rr-Rs); %

if dis < .5
    HT(1) = 1;
    return;
end;

N1 = floor(NPTS/(RL(1)*2))+1; % Range of sum
N2 = floor(NPTS/(RL(2)*2))+1; %
```

```
N3 = floor(NPTS/(RL(3)*2))+1;

for NX = -N1:N1
    for NY = -N2:N2
        for NZ = -N3:N3
            NR = [NX;NY;NZ];
            DELP = LTHIMAGE(Rr,Rs,RL,NR,PERM);
            for i = 1:8
                ID = round(DELP(i));
                FDM1 = ID;
                ID = ID + 1;
                if ID > NPTS
                    break;
                end;
                %
                GID = BETA(1,:)'.^abs(NR-PERM2(:,i)).*BETA(2,:)'.^abs(NR);
                GID = GID(1)*GID(2)*GID(3)/FDM1;
                HT(ID) = HT(ID) + GID;
            end;
        end;
    end;
end;


% Filter with Hi Pass filter of 1% of sampling freq (i.e. 100Hz)
W = 2.*4.*atan(1.)*100.;          %
T = .0001;                        %
R1= exp(-W*T); R2= R1;            %
B1= 2.*R1*cos(W*T); B2= -R1^2;    %
A1= -(1.+R2); A2= R2;             %
Y1= 0; Y2= 0; Y0= 0;

for i = 1:NPTS % Filter HT
    X0 = HT(i);
    HT(i) = Y0 + A1*Y1 + A2*Y2;
    Y2 = Y1;
    Y1 = Y0;
    Y0 = B1*Y1 + B2*Y2 + X0;
end;
```

lthimage.m

```
function DELP = LTHIMAGE(DRr,DRs,RL,NR,PERM)
%***********************************************************************
% Subroutine to calculate a room impulse response.
```

```
%
% Input Arguments:
%   Name: DRr
%   Type: 3*1 real vector
%   Description: Vector radius to receiver in smaple preiods = length/(C*T)
%
%   Name: DRs
%   Type: 3*1 real vector
%   Description: Vector radius to source in smaple preiods = length/(C*T)
%
%   Name: RL
%   Type: 3*1 real vector
%   Description: Vector of box dimensions in smaple preiods = length/(C*T)
%
%   Name: NR
%   Type: 3*1 real vector
%   Description: Vector of mean image number
%
% Output Arguments:
%   Name: DELP
%   Type: 8*1 real vector
%   Description: Vector of 8 source to images distances in smaple periods
%
% AUTHORS: Ning Chen, Dr. De Leon
% Version: 1.0
% Creation Date: Jul. 31, 2001
% Last Revision:
% References:
%*******************************************************************

for i = 1:8
    RP(:,i) = DRr + PERM(:,i).*DRs;
end;

R2L = 2*RL.*NR; for i = 1:8
    R1 = R2L - RP(:,i);
    DELP(i) = norm(R1);
end;
```

siri_meas1.m

```
function SIRI = siri_meas1(T, U, s1, s2, H, W);
%----------------------------------------------------------------------
% SIRI MEASUREMENT 1 (With known sources and mixing filters)
```

```
%
% This MATLAB code determine the SIRI with known sources and mixing filters.
%
% Call Syntax: SIRI = siri_meas1(T,U,s1,s2,H,W)
%
% INPUT arguments:
% Name: T
% Type: scalar
% Description: Block length
%
% Name: U
% Type: scalar
% Description: Number of blocks
%
% INPUT arguments:
% Name: s1/2
% Type: T*U matrix
% Description: Short-Time Fourier representative of source signals
%
% Name: H
% Type: 2*2*T matrix
% Description: Mixing filter matrix
%
% Name: W
% Type: 2*2*T matrix
% Description: Un-mixing filter matrix
%
% OUTPUT arguments:
% Name: SIRI
% Type: scalar
% Description: Signal-to-Interference Ratio Improvement
%
% Creation Date: Aug. 06, 2001
% Last Revision: Sept. 10, 2001
%-----------------------------------------------------------------

%-----------------------------------------------------------------

% The system overall filter
A=zeros(2,2,T); for omega = 1:T
  A(:,:,omega) = W(:,:,omega) * H(:,:,omega);
end;

%-----------------------------------------------------------------
% Power of the filters
```

```
A11=abs(reshape(A(1,1,:),T,1)).^2;
A12=abs(reshape(A(1,2,:),T,1)).^2;
A21=abs(reshape(A(2,1,:),T,1)).^2;
A22=abs(reshape(A(2,2,:),T,1)).^2; %
HH11=abs(H11).^2; HH12=abs(H12).^2;%
HH21=abs(H21).^2; HH22=abs(H22).^2;

%-------------------------------------------------------------------------
% SNRI
num = 0; num1 = 0; den1 = 0; den = 0; %
for m = 1:U
   sf1 = abs(s1(:,m)).^2;  sf2 = abs(s2(:,m)).^2;
   num1= num1 + sum(HH11.*sf1) + sum(HH22.*sf2);
   den1= den1 + sum(HH12.*sf2) + sum(HH21.*sf1);
   num = num  + sum(A11.*sf1) + sum(A22.*sf2);
   den = den  + sum(A21.*sf1) + sum(A12.*sf2);
end;

SIRi = 10*log10(num1/den1); % Input SIR
SIRo = 10*log10(num/den);   % Output SIR
SIRI = SIRo - SIRi;         % SIR Improvement
```

siri_meas2.m

```
function SIRI = siri_meas2(T, U, x, W);
%-------------------------------------------------------------------------
% SIRI MEASUREMENT 2 (Without known sources and mixing filters)
%
% This MATLAB code determine the SIRI without known sources and mixing filters.
%
% Call Syntax: SIRI = siri_meas2(T,U,W)
%
% INPUT arguments:
% Name: T
% Type: scalar
% Description: Block length
%
% Name: U
% Type: scalar
% Description: Number of blocks
%
% Name: x
% Type: T*U*4 matrix
% Description: Short-Time Fourier representative of mixed signals
%
% Name: W
% Type: 2*2*T matrix
% Description: Un-mixing filter matrix
%
% OUTPUT arguments:
% Name: SIRI
% Type: scalar
% Description: Signal-to-Interference Ratio Improvement
%
% Creation Date: Aug. 06, 2001
% Last Revision: Sept. 10, 2001
%-----------------------------------------------------------------------

%-----------------------------------------------------------------------
% Un-mixing filters
W11 = reshape(W(1,1,:),T,1); W12 = reshape(W(1,2,:),T,1); %
W21 = reshape(W(2,1,:),T,1); W22 = reshape(W(2,2,:),T,1);

%-----------------------------------------------------------------------
% SNRI
num1 = 0; den1 = 0; num=0; den=0; %
for m = 1:U
  num1= num1 + sum(abs(x(:,m,1)).^2) + sum(abs(x(:,m,4)).^2);  %% Calculate...
  den1= den1 + sum(abs(x(:,m,2)).^2) + sum(abs(x(:,m,3)).^2);  %% SIRi
  num = num  + sum(abs(W11.*x(:,m,1)+W12.*x(:,m,3)).^2) + ...
               sum(abs(W21.*x(:,m,2)+W22.*x(:,m,4)).^2);
  den = den  + sum(abs(W11.*x(:,m,2)+W12.*x(:,m,4)).^2) + ...
               sum(abs(W21.*x(:,m,1)+W22.*x(:,m,3)).^2);
end;

SIRi = 10*log10(num1/den1); % Signal to Interference Ratio of the input
SIRo = 10*log10(num/den);   % Signal to Interference Ratio of the output
SIRI = SIRo - SIRi;         % SIR Improvement
```

Appendix C - C Simulation Codes

speech.c

```
/* -------------------------------------------------------------------------------
 * CO-CHANNEL CONVOLUTIVE MIXED SPEECH SEPARATOR

 * This C code implements the blind speech separation algorithm of Ikram
 * and Morgan [1].  The problem is as follows.  Given two signals, x1 and
 * x2 each containing a convolutive mixture of two source speech signals, s1 and
 * s2 produce separated speech signals, y1 and y2 such that y1 ~ s1, y2 ~ s2.
 * This is a blind separation problem since we only have x1 and x2 and no
 * other additional information.  The approach assumes the uncorrelatedness
 * between the original source signals.  We then attempt to diagonalize the
 * estimated Power Spectral Density (PSD) matrices at multiple time segments to
 * decorrelate the mixtures in frequency-domain (avoiding the deconvolution in
 * time-domain) to achieve the separation.
 * -------------------------------------------------------------------------------
 * SYNTAX
 * -------------------------------------------------------------------------------
 * From command line: ./speech <file1> <file2>
 *
 * Where <fileX> are the files containing the the two mixtures.
 * -------------------------------------------------------------------------------
 * NOTES
 * -------------------------------------------------------------------------------
 * 1) This version of the code is NOT optimized for performance, it is written
 *    now for readability and mathematical correctness.  The reader may notice
 *    lines of code that do something that will not change the numerical value
 *    of whats being calculated but it is what would happen if you carried out
 *    the matrix calculation.  These lines of code can easily be removed for
 *    performance at a later time but are left in to account for correctness.
 *
 * 2) Everything that deals with the vectors WXX_final can be removed for
 *    performance increase, the values are calulated purely for end user
 *    information,
 *
 * 3) The code currently accepts input of ASCII files only, this is for ease of
 *    use.  The input and output values can easily and quickly be evaluated.
 *    For performance later, switching to binary inputs may be desired.  A
 *    program such as Matlab can be used to take and put back the ASCII files
 *    into .wav formats.
 *
 * 4) There are portions of the code that will need to be modified for a
 *    real-time implementation as we will only have blocks of samples to work on
*    and not the entire signal.  Control mechanisms will will need to be added
*    as well to control what block is being operated on and what block is
*    being filled with new samples.
*
* 5) Output will be stored in output_source1.txt and output_source2.txt.  These
*    can be modified as desired.
*
* 6) The only variable that needs to be changed from mixture to mixture is
*    SIGNAL_LENGTH unless other parameters want to be changed, i.e the number of
*    stages or the iterations.
*
* 7) The code is fairly straight forward if you have a copy of the Matlab
*    version as well, so it is recommended that the reader review the Matlab
*    verson of the code as it is easier to understand.
*
* 8) This code was developed on a Linux box running the 2.4.x kernel using libs
*    from glibc-2.2.4.  It was compiled using gcc-2.96.  It adheres to ANSI
*    standard so it should be easily portable.
* -----------------------------------------------------------------------------
* REFERENCES
* -----------------------------------------------------------------------------
* [1] M. Ikram and D. Morgan, "A multiresolution Approach to Blind Separation of
*     Speech Signals in A Reverberant Environment", In Proc. IEEE ICASSP, Pages
*     2757-2760, 2001.
* [2] L. Parra and C. Spence, "Convolutive Blind Separation of Non-Stationary
*     Sources", IEEE Transactions on Speech and Audio Processing, 8(3):320-327,
*     May 2000.
*
* -----------------------------------------------------------------------------
* VERSIONS
* -----------------------------------------------------------------------------
* 1.2.1 (12/18/01) Last minute comment changes.
*
* 1.2.0 (12/17/01) Code now matches latest algorithm implementation and is
*                  working.
*                  Add normalization of output vectors.
*                  Cleanup variables and memory allocations.
*                  Lots of commenting.
*
* 1.1.0 (12/01/01) Start merge to final version of algorithm using the final
*                  Matlab code as a blueprint.
*                  Put the 2 stages of the multistage iteration into a for loop.
*                  Change how Frobenius norm is calculated.
*                  Apply Hermitian property to Rx calculations.
*
```

```
 * 1.0.1 (11/28/01) Cleanup code a little.
 *
 * 1.0.0 (11/18/01) Working code.  Still missing normalization code, but is not
 *                  that important.
 *                  Output samples match Matlab output to the 15/16 digit after
 *                  normalization in Matlab.
 *
 * 0.9.8 (11/18/01) Make a dissmal attempt at normalization code.
 *                  Finish stage 2 of multistage iteration.
 *
 * 0.9.7 (11/11/01) Add code to rebuild the output vectors from blocks.
 *
 * 0.9.6 (10/22/01) Finish code to calculate cost function.
 *                  Finish code to calculate gradient.
 *                  Add code to calculate seperation matrix.
 *                  Add code to apply the symmetric constraint on the seperation
 *                  matrix.
 *                  Add truncation of seperation matrix in time-domain.
 *                  Finish stage 1 of multistage iteration.
 *
 * 0.9.5 (09/28/01) Add code to calculate Frobenius norm.
 *                  Start addition of code to calculate cost function.
 *                  Start addition of code to calculate gradient.
 *
 * 0.9.4 (07/14/01) Add covariance calculation code.
 *
 * 0.9.3 (07/03/01) Improved implementation of Short Time Fourier block of code.
 *
 * 0.9.2 (06/19/01) Add Short Time Fourier Transform block of code.
 *
 * 0.9.1 (06/18/01) Link in complex math code(fft, ifft) and their dependencies.
 *                  Construct hamming window calculation.
 *
 * 0.9.0 (06/17/01) Initial code, setup up I/O, and command line parameters.
 * ---------------------------------------------------------------------------
 * CONTACT
 * ---------------------------------------------------------------------------
 * Bug Reports
 * Report all bugs to Prof. Phillip De Leon (pdeleon@nmsu.edu)
 *----------------------------------------------------------------------------
*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "cmplx.h"
```

```
void fft();
void ifft();

int main(int argc, char *argv[])
{
    FILE *input_file1, *input_file2;
    FILE *output_file1, *output_file2;

    /* Set initial parameters */
    const double PI = 3.1415926535897932;  /* Value of pi                   */
    const double BETA = 0.5;               /* Overlapping factor            */
    const double ALPHA = 1;                /* Normalized step size          */
    const double E = 0.0000001;            /* Offset used in normalization */

    const int SIGNAL_LENGTH = 409600;    /* Signal length in samples        */
    const int BLOCK_LENGTH = 2048;       /* Basic block length              */
    const int ITERATIONS = 100;          /* Number of iterations            */
    const int NUMBER_SUPERBLOCKS = 6;    /* The total number of superblocks */
    const int NUM_FILTER_STAGES = 2;     /* Multistage parameters           */
    /* Half the value of block length,  avoids doing multiple divides */
    const int HALF_BLOCK_LENGTH = (BLOCK_LENGTH / 2);
    /* Number of blocks                      */
    const int NUMBER_BLOCKS = floor(SIGNAL_LENGTH / (BETA * BLOCK_LENGTH) - 1);
    /* Number of blocks in each superblock */
    const int NUMBER_BLOCKS_IN_SUPERBLOCK =
      floor(NUMBER_BLOCKS / NUMBER_SUPERBLOCKS);
    /* Length of output signal */
    const int OUTPUT_SIGNAL_LENGTH = (SIGNAL_LENGTH + (BLOCK_LENGTH -
                                     (BETA * BLOCK_LENGTH)));
    /* Calculate once to avoid many divides later */
    const double INV_NUMBER_BLOCKS_IN_SUPERBLOCK =
                (1 / (double)NUMBER_BLOCKS_IN_SUPERBLOCK);

    int i = 0;                 /* Bogus variable for loops                       */
    int j = 0;                 /* Bogus variable for loops                       */
    int k = 0;                 /* Bogus variable for loops                       */
    int w = 0;                 /* Little omega                                   */
    int q = 0;                 /* Current iteration of stage                     */
    int xx_index = 0;          /* Initialize index for xx vectors                */
    int xw_index = 0;          /* Initialize index for xw vectors                */
    int conjugate_index = 0; /* Initialize index for symmetric constraint
      * calculation                                      */
    int block = 0;             /* Initialize block number                        */
    int y_counter = 0;         /* Initialize counter for y vectors               */
    int yy_counter = 0;      /* Initialize counter for yy vectors               */
    int L1 = 2048; /* The length of y1 and y2, used in reassembling the blocks */
    int Q[2] = {500, 2048}; /* q values for multistage iteration              */
    double max_sample_value_1 = 0;  /* Maximum value of sample, used in
     * normalization                                 */
    double max_sample_value_2 = 0;  /* Maximum value of sample, used in
     * normalization                                 */

    /* Initialization for "double" vectors (All pointers) */
    double *omega;
    double *xt1;
    double *xt2;
    /* Initialization for "complex" values */
    complex fronorm = cmplx(0.0,0.0);
    complex gradt11; complex gradt12; complex gradt21; complex gradt22;
    complex V11; complex V12; complex V21; complex V22;
    /* Initialization for "complex" vectors (All pointers)*/
    complex *mu;
    complex *xw1; complex *xw2;
    complex *xx11; complex *xx12; complex *xx21; complex *xx22;
    /* Pointer to a pointer for RxXX*/
    complex **Rx11; complex **Rx12; complex **Rx21; complex **Rx22;
    complex *Ws_11; complex *Ws_12; complex *Ws_21; complex *Ws_22;
    complex *W11; complex *W12; complex *W21; complex *W22;
    complex *W11_temp; complex *W12_temp; complex *W21_temp; complex *W22_temp;
    complex *W11_final; complex *W12_final;
    complex *W21_final; complex *W22_final;
    complex *Y1; complex *Y2;
    complex *yy1; complex *yy2;
    complex *y1; complex *y2;

    complex *x1[BLOCK_LENGTH];  /* Static array in one dimension, dynamic in
 * other */
    complex *x2[BLOCK_LENGTH];  /* Static array in one dimension, dynamic in
 * other */

    /* Check to see if we have the correct number of
       arguments from command line at startup          */
    if (argc != 3) {
fprintf(stderr, "Syntax: speech [file1] [file2]\n");
exit(8);
    }
    /* Open our input files */
    if ((input_file1 = fopen(argv[1], "r")) == NULL) {
fprintf(stderr, "speech: Cannot open %s\n", argv[1]);
exit(8);
    }
    if ((input_file2 = fopen(argv[2], "r")) == NULL) {
fprintf(stderr, "speech: Cannot open %s\n", argv[2]);
exit(8);
    }
    /* Reserve memory spaces for vectors omega, xt1, xt2, xw1, xw2, and Ws_XX */
    if ((omega = calloc((BLOCK_LENGTH), sizeof(double))) == NULL) {
fprintf(stderr, "Out of memory\n");
exit(8);
    }
    if ((xt1 = calloc(SIGNAL_LENGTH, sizeof(double))) == NULL) {
fprintf(stderr, "Out of memory\n");
exit(8);
    }
    if ((xt2 = calloc(SIGNAL_LENGTH, sizeof(double))) == NULL) {
fprintf(stderr, "Out of memory\n");
exit(8);
    }
    if ((xw1 = calloc(BLOCK_LENGTH, sizeof(complex))) == NULL) {
fprintf(stderr, "Out of memory\n");
exit(8);
    }
    if ((xw2 = calloc(BLOCK_LENGTH, sizeof(complex))) == NULL) {
fprintf(stderr, "Out of memory\n");
exit(8);
    }
    if ((Ws_11 = malloc(BLOCK_LENGTH * sizeof(complex))) == NULL) {
fprintf(stderr, "Out of memory\n");
exit(8);
    }
    if ((Ws_12 = malloc(BLOCK_LENGTH * sizeof(complex))) == NULL) {
fprintf(stderr, "Out of memory\n");
exit(8);
    }
    if ((Ws_21 = malloc(BLOCK_LENGTH * sizeof(complex))) == NULL) {
fprintf(stderr, "Out of memory\n");
exit(8);
    }
    if ((Ws_22 = malloc(BLOCK_LENGTH * sizeof(complex))) == NULL) {
fprintf(stderr, "Out of memory\n");
exit(8);
    }
    /* The four callocs below can be removed for performance,
     * is purely for user information                       */
    if ((W11_final = calloc(BLOCK_LENGTH, sizeof(complex))) == NULL) {
        fprintf(stderr, "Out of memory\n");
exit(8);
    }
    if ((W12_final = calloc(BLOCK_LENGTH, sizeof(complex))) == NULL) {
fprintf(stderr, "Out of memory\n");
exit(8);
    }
    if ((W21_final = calloc(BLOCK_LENGTH, sizeof(complex))) == NULL) {
fprintf(stderr, "Out of memory\n");
exit(8);
    }
    if ((W22_final = calloc(BLOCK_LENGTH, sizeof(complex))) == NULL) {
fprintf(stderr, "Out of memory\n");
exit(8);
    }

    for (i = 0; i < BLOCK_LENGTH; i++) { /*This can be removed for performance,*/
W11_final[i] = cmplx(1.0,0.0);   /* is purely for user information    */
W12_final[i] = cmplx(0.0,0.0);  /* Initialize the vectors that       */
W21_final[i] = cmplx(0.0,0.0);  /* store the final seperation matrix */
W22_final[i] = cmplx(1.0,0.0);   /*                                   */
    }

    /* Create storage space for a BLOCK_LENGTH * NUMBER_BLOCKS
     *atrix of matrices for x1 and x2                          */
    for(i = 0; i < BLOCK_LENGTH; i++) {
if ((x1[i] = calloc(NUMBER_BLOCKS, sizeof(complex))) == NULL) {
    fprintf(stderr, "Out of memory\n");
    exit(8);
}
if ((x2[i] = calloc(NUMBER_BLOCKS, sizeof(complex))) == NULL) {
    fprintf(stderr, "Out of memory\n");
    exit(8);
}
    }
    /* Read mixtures into vectors */
    i = 0;
    while ((fscanf(input_file1, "%lf", &xt1[i])) &&
           (fscanf(input_file2, "%lf", &xt2[i])) != EOF) {
i++;
    }

    /* Close input files */
    fclose(input_file1);
    fclose(input_file2);

    fprintf(stderr,"Starting Stages\n");
    /*-------------------------------------------------------------------------*/
    /* Hamming window calculations */
    for(i = 0; i < BLOCK_LENGTH; i++) {
omega[i] = 0.54 - 0.46 * cos(2 * PI * i / (BLOCK_LENGTH - 1));
    }
    /*-------------------------------------------------------------------------*/
    /* Short-Time Fourier Transform */
    for(i = 0; i < NUMBER_BLOCKS; i++) {
block = (BETA * BLOCK_LENGTH * i);  /* Calculate what block we are on  */
xw_index = 0;
        for(j = block; j < (block + BLOCK_LENGTH); j++) {
    /* The column vector of windowed mixtures (channel 1) */
    xw1[xw_index] = cmplx(omega[xw_index] * xt1[j],0);
    /* The column vector of windowed mixtures (Channel 2) */
    xw2[xw_index] = cmplx(omega[xw_index] * xt2[j],0);
    xw_index++;
        }
fft(BLOCK_LENGTH, xw1);  /* Fast Fourier Transform */
fft(BLOCK_LENGTH, xw2);  /* Fast Fourier Transform */
/* Store FFT values in a T x NUMBER_BLOCKS matrix */
for(k = 0; k < BLOCK_LENGTH; k++) {
    /* Store calculated xw1 into our BLOCK_LENGTH*NUMBER_BLOCKS matrix */
    x1[k][i] = xw1[k];
    /* Store calculated xw2 into our BLOCK_LENGTH*NUMBER_BLOCKS matrix */
    x2[k][i] = xw2[k];
}
    }
    /* Cleanup any storage we don't need anymore */
    free(xt1); free(xt2); free(omega); free(xw1); free(xw2);
    /*---------------------------------------------- -------------------------*/
    /* MULTISTAGE ITERATION  */
    /*-------------------------------------------------------------------------*/
    for (q = 0; q < NUM_FILTER_STAGES; q++) {
fprintf(stderr,"Entering Stage %d\n", q);
/* Allocate space for RxXX vectors */
if ((Rx11 = calloc(NUMBER_SUPERBLOCKS, sizeof(complex *))) == NULL) {
    fprintf(stderr, "Out of memory\n");
    exit(8);
}
if ((Rx12 = calloc(NUMBER_SUPERBLOCKS, sizeof(complex *))) == NULL) {
    fprintf(stderr, "Out of memory\n");
    exit(8);
```

```
}
if ((Rx21 = calloc(NUMBER_SUPERBLOCKS, sizeof(complex *))) == NULL) {
    fprintf(stderr, "Out of memory\n");
    exit(8);
}
if ((Rx22 = calloc(NUMBER_SUPERBLOCKS, sizeof(complex *))) == NULL) {
    fprintf(stderr, "Out of memory\n");
    exit(8);
}
for(i = 0; i < NUMBER_SUPERBLOCKS; i++) {
    /* Allocate space for xxXX vectors */
    if ((xx11 = calloc(HALF_BLOCK_LENGTH + 1, sizeof(complex))) == NULL){
fprintf(stderr, "Out of memory\n");
exit(8);
    }
    if ((xx12 = calloc(HALF_BLOCK_LENGTH + 1, sizeof(complex))) == NULL){
fprintf(stderr, "Out of memory\n");
exit(8);
    }
    if ((xx21 = calloc(HALF_BLOCK_LENGTH + 1, sizeof(complex))) == NULL){
fprintf(stderr, "Out of memory\n");
exit(8);
    }
    if ((xx22 = calloc(HALF_BLOCK_LENGTH + 1, sizeof(complex))) == NULL){
fprintf(stderr, "Out of memory\n");
exit(8);
    }
    for(j = 0; j < NUMBER_BLOCKS_IN_SUPERBLOCK; j++) {
/* Calculate where we are in the super block */
xx_index = (NUMBER_BLOCKS_IN_SUPERBLOCK * i) + j;
/*Computation of covariance*/
for(k = 0; k < (HALF_BLOCK_LENGTH + 1); k++) {
    xx11[k] = cadd(xx11[k],(cmul(x1[k][xx_index],
                            (conjg(x1[k][xx_index])))));
    xx12[k] = cadd(xx12[k],(cmul(x1[k][xx_index],
                            (conjg(x2[k][xx_index])))));
    xx21[k] = conjg(xx12[k]); /* Hermitian property of
       * PSD matrices          */
    xx22[k] = cadd(xx22[k],(cmul(x2[k][xx_index],
                            (conjg(x2[k][xx_index])))));
}
    }
    /* Multiply all the values in xx vectors
            * by 1 / NUMBER_BLOCKS_IN_SUPERBLOCK    */
    for(k = 0; k < (HALF_BLOCK_LENGTH + 1); k++) {
```

```
xx11[k] = cmul(xx11[k],cmplx(INV_NUMBER_BLOCKS_IN_SUPERBLOCK,0));
xx12[k] = cmul(xx12[k],cmplx(INV_NUMBER_BLOCKS_IN_SUPERBLOCK,0));
xx21[k] = cmul(xx21[k],cmplx(INV_NUMBER_BLOCKS_IN_SUPERBLOCK,0));
xx22[k] = cmul(xx22[k],cmplx(INV_NUMBER_BLOCKS_IN_SUPERBLOCK,0));
    }
    /* Save the xx vectors calculated above, NUMBER_SUPERBLOCKS of them
     * should be stored */
    Rx11[i] = xx11;
    Rx12[i] = xx12;
    Rx21[i] = xx21;
    Rx22[i] = xx22;
}

/* Allocate space for seperation matrix */
if ((W11 = calloc(BLOCK_LENGTH, sizeof(complex))) == NULL) {
    fprintf(stderr, "Out of memory\n");
    exit(8);
}
if ((W12 = calloc(BLOCK_LENGTH, sizeof(complex))) == NULL) {
    fprintf(stderr, "Out of memory\n");
    exit(8);
}
if ((W21 = calloc(BLOCK_LENGTH, sizeof(complex))) == NULL) {
    fprintf(stderr, "Out of memory\n");
    exit(8);
}
if ((W22 = calloc(BLOCK_LENGTH, sizeof(complex))) == NULL) {
    fprintf(stderr, "Out of memory\n");
    exit(8);
}
if ((mu = malloc((HALF_BLOCK_LENGTH + 1) * sizeof(complex))) == NULL) {
    fprintf(stderr, "Out of memory\n");
    exit(8);
}
/* Average power of the mixtures at each frequency */
for (j = 0; j < (HALF_BLOCK_LENGTH + 1); j ++) {
    /* Initialize Frobenius norm to zero for each stage */
    fronorm = cmplx(0.0,0.0);
    /* Calculate the Frobenius norm      */
    /* Rx11^2 + Rx12^2 + Rx21^2 + Rx22^2 */
    for (k = 0; k < NUMBER_SUPERBLOCKS; k++) {
fronorm = cadd(fronorm,
        cadd(cmul((*(Rx11+k))[j],conjg((*(Rx11+k))[j])),
                cadd(cmul((*(Rx12+k))[j],conjg((*(Rx12+k))[j])),
        cadd(cmul((*(Rx21+k))[j],conjg((*(Rx21+k))[j])),
        cmul((*(Rx22+k))[j],conjg((*(Rx22+k))[j]))))));
    }
    /* Divide the step size ALPHA by the calculated Frobenius norm
     * (normalize)*/
    mu[j] = cdiv(cmplx(ALPHA,0.0),fronorm);
}
/* Reinitializing the un-mixing filter matrices at each stage */
/* Initialize un-mixing filter matrices to identiy matrix */
for(j = 0; j < BLOCK_LENGTH; j++) {
    Ws_11[j] = cmplx(1.0,0.0);
    Ws_12[j] = cmplx(0.0,0.0);
    Ws_21[j] = cmplx(0.0,0.0);
    Ws_22[j] = cmplx(1.0,0.0);
}

for(j = 0; j < ITERATIONS; j++) {  /* Iterations for ITERATIONS times */
     /* Seek W(omega) at every frequency bin */
    for (w = 0; w < (HALF_BLOCK_LENGTH + 1); w++) {
gradt11 = cmplx(0.0,0.0); /*                             */
gradt12 = cmplx(0.0,0.0); /* Initialize Gradient to zero */
gradt21 = cmplx(0.0,0.0); /*                             */
gradt22 = cmplx(0.0,0.0); /*                             */
for(k = 0; k < NUMBER_SUPERBLOCKS; k++) {
    V11 = cadd(cmul(conjg(Ws_11[w]),cadd(cmul(Ws_11[w],
                           (*(Rx11+k))[w]),cmul(Ws_12[w],(*(Rx21+k))[w]))),
                           cmul(conjg(Ws_12[w]),cadd(cmul(Ws_11[w],
                           (*(Rx12+k))[w]),cmul(Ws_12[w],(*(Rx22+k))[w]))));
    V12 = cadd(cmul(conjg(Ws_21[w]),cadd(cmul(Ws_11[w],
                          (*(Rx11+k))[w]),cmul(Ws_12[w],(*(Rx21+k))[w]))),
                          cmul(conjg(Ws_22[w]),cadd(cmul(Ws_11[w],
                          (*(Rx12+k))[w]),cmul(Ws_12[w],(*(Rx22+k))[w]))));
    V21 = cadd(cmul(conjg(Ws_11[w]),cadd(cmul(Ws_21[w],
                          (*(Rx11+k))[w]),cmul(Ws_22[w],(*(Rx21+k))[w]))),
                          cmul(conjg(Ws_12[w]),cadd(cmul(Ws_21[w],
                          (*(Rx12+k))[w]),cmul(Ws_22[w],(*(Rx22+k))[w]))));
    V22 = cadd(cmul(conjg(Ws_21[w]),cadd(cmul(Ws_21[w],
                          (*(Rx11+k))[w]),cmul(Ws_22[w],(*(Rx21+k))[w]))),
                          cmul(conjg(Ws_22[w]),cadd(cmul(Ws_21[w],
                          (*(Rx12+k))[w]),cmul(Ws_22[w],(*(Rx22+k))[w]))));
    V11 = csub(V11,V11);              /*                         */
    V12 = csub(V12,cmplx(0.0,0.0));  /* Cost function            */
    V21 = csub(V21,cmplx(0.0,0.0));  /* (can be made much more */
    V22 = csub(V22,V22);              /*  effiecient)            */
    gradt11 = cadd(gradt11,cadd(cmul((*(Rx11+k))[w],
                         cadd(cmul(V11,Ws_11[w]),cmul(V12,Ws_21[w]))),
                                       cmul((*(Rx21+k))[w],cadd(cmul(V11,Ws_12[w]),
                                       cmul(V12,Ws_22[w])))));
    gradt12 = cadd(gradt12,cadd(cmul((*(Rx12+k))[w],
                                       cadd(cmul(V11,Ws_11[w]),cmul(V12,Ws_21[w]))),
                                       cmul((*(Rx22+k))[w],cadd(cmul(V11,Ws_12[w]),
                                       cmul(V12,Ws_22[w])))));
    gradt21 = cadd(gradt21,cadd(cmul((*(Rx11+k))[w],
                                       cadd(cmul(V21,Ws_11[w]),cmul(V22,Ws_21[w]))),
                                       cmul((*(Rx21+k))[w],cadd(cmul(V21,Ws_12[w]),
                                       cmul(V22,Ws_22[w])))));
    gradt22 = cadd(gradt22,cadd(cmul((*(Rx12+k))[w],
                                       cadd(cmul(V21,Ws_11[w]),cmul(V22,Ws_21[w]))),
                                       cmul((*(Rx22+k))[w],cadd(cmul(V21,Ws_12[w]),
                                       cmul(V22,Ws_22[w])))));
}
/* Gradient(can be made more efficient) */
gradt11 = cmul(cmplx(2.0,0.0),csub(gradt11,gradt11));
gradt12 = cmul(cmplx(2.0,0.0),csub(gradt12,cmplx(0.0,0.0)));
gradt21 = cmul(cmplx(2.0,0.0),csub(gradt21,cmplx(0.0,0.0)));
gradt22 = cmul(cmplx(2.0,0.0),csub(gradt22,gradt22));
Ws_11[w] = csub(Ws_11[w],cmul(mu[w],gradt11));  /*        */
Ws_12[w] = csub(Ws_12[w],cmul(mu[w],gradt12));  /* Update */
Ws_21[w] = csub(Ws_21[w],cmul(mu[w],gradt21));  /*        */
Ws_22[w] = csub(Ws_22[w],cmul(mu[w],gradt22));  /*        */
    }
    /* Apply Symmetric constraint */
    conjugate_index = 0;
    for(i = HALF_BLOCK_LENGTH + 1; i < BLOCK_LENGTH; i++) {
Ws_11[i] = conjg(Ws_11[HALF_BLOCK_LENGTH - conjugate_index - 1]);
Ws_12[i] = conjg(Ws_12[HALF_BLOCK_LENGTH - conjugate_index - 1]);
Ws_21[i] = conjg(Ws_21[HALF_BLOCK_LENGTH - conjugate_index - 1]);
Ws_22[i] = conjg(Ws_22[HALF_BLOCK_LENGTH - conjugate_index - 1]);
conjugate_index++;
    }
    /* Truncation of W(omega) in time-domain */
    for (i = 0; i < BLOCK_LENGTH; i++) {
W11[i] = Ws_11[i];
W12[i] = Ws_12[i];
W21[i] = Ws_21[i];
W22[i] = Ws_22[i];
    }
    ifft(BLOCK_LENGTH,W11);  /*                          */
    ifft(BLOCK_LENGTH,W12);  /* Transfer to time domain */
    ifft(BLOCK_LENGTH,W21);  /*                          */
    ifft(BLOCK_LENGTH,W22);  /*                          */
    /* Needed to zero pad the q length block with BLOCK_LENGTH-q zeros */
    if ((W11_temp = calloc(BLOCK_LENGTH, sizeof(complex))) == NULL) {
fprintf(stderr, "Out of memory\n");
exit(8);
    }
    if ((W12_temp = calloc(BLOCK_LENGTH, sizeof(complex))) == NULL) {
fprintf(stderr, "Out of memory\n");
exit(8);
    }
    if ((W21_temp = calloc(BLOCK_LENGTH, sizeof(complex))) == NULL) {
fprintf(stderr, "Out of memory\n");
exit(8);
    }
    if ((W22_temp = calloc(BLOCK_LENGTH, sizeof(complex))) == NULL) {
fprintf(stderr, "Out of memory\n");
exit(8);
    }

    /* Copy the first Q[q] samples over to the temporary vectors */
    for(i = 0; i < Q[q]; i++) {
W11_temp[i] = W11[i];
W12_temp[i] = W12[i];
W21_temp[i] = W21[i];
W22_temp[i] = W22[i];
    }

    fft(BLOCK_LENGTH, W11_temp);  /*                                   */
    fft(BLOCK_LENGTH, W12_temp);  /* Transfer back to frequency domain */
    fft(BLOCK_LENGTH, W21_temp);  /*                                   */
    fft(BLOCK_LENGTH, W22_temp);  /*                                   */

    for (i = 0; i < BLOCK_LENGTH; i++) {
Ws_11[i] = W11_temp[i];  /*                               */
Ws_12[i] = W12_temp[i];  /* Save values to be used again in */
Ws_21[i] = W21_temp[i];  /* next stage.                   */
Ws_22[i] = W22_temp[i];  /*                               */
    }
}
/* End of ITERATIONS iterations */

/* Allocate space */
if ((Y1 = calloc(BLOCK_LENGTH, sizeof(complex))) == NULL) {
    fprintf(stderr, "Out of memory\n");
    exit(8);
```

```
}
if ((Y2 = calloc(BLOCK_LENGTH, sizeof(complex))) == NULL) {
    fprintf(stderr, "Out of memory\n");
    exit(8);
}

/* Current Stage Seperation */
for (i = 0; i < NUMBER_BLOCKS; i++) {
    for(j = 0; j < BLOCK_LENGTH; j++) {
/* Multiply the current seperation matrix by input signals */
Y1[j] = cadd(cmul(Ws_11[j],x1[j][i]),cmul(Ws_12[j],x2[j][i]));
/* Multiply the current seperation matrix by input signals */
Y2[j] = cadd(cmul(Ws_21[j],x1[j][i]),cmul(Ws_22[j],x2[j][i]));
x1[j][i] = Y1[j];  /* Save seperated signal for next stage */
x2[j][i] = Y2[j];  /* Save seperated signal for next stage */
    }
}
/* Update the overall un-mixing filter matrices (This can be removed for
 * performance, it is purely user information)*/
for (i = 0; i < BLOCK_LENGTH; i++) {
    W11_final[i] = cadd(cmul(Ws_11[i],W11_final[i]),
                        cmul(Ws_12[i],W21_final[i]));
    W12_final[i] = cadd(cmul(Ws_11[i],W12_final[i]),
                        cmul(Ws_12[i],W22_final[i]));
    W21_final[i] = cadd(cmul(Ws_21[i],W11_final[i]),
                        cmul(Ws_22[i],W21_final[i]));
    W22_final[i] = cadd(cmul(Ws_21[i],W12_final[i]),
                        cmul(Ws_22[i],W22_final[i]));
}

/* Cleanup all the storage vectors that are no longer needed */
free(xx11); free(xx12); free(xx21); free(xx22);
free(Rx11); free(Rx12); free(Rx21); free(Rx22);
free(W11); free(W12); free(W21); free(W22);
free(W11_temp); free(W12_temp); free(W21_temp); free(W22_temp);
free(Y1); free(Y2);
free(mu);
fprintf(stderr, "Leaving Stage %d\n", q);
    }

    fprintf(stderr, "Finished Stages\n");
    fprintf(stderr, "Seperation Complete\n");
    /*---------------------  ---------------------------------------------*/
    /* END of MULTISTAGE ITERATION */
    /*------------------------------------------------------------------*/
    /* Allocate space */
    if ((Y1 = calloc(BLOCK_LENGTH, sizeof(complex))) == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(8);
    }
    if ((Y2 = calloc(BLOCK_LENGTH, sizeof(complex))) == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(8);
    }
    if ((yy1 = calloc(BLOCK_LENGTH, sizeof(complex))) == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(8);
    }
    if ((yy2 = calloc(BLOCK_LENGTH, sizeof(complex))) == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(8);
    }
    /* Create storage space for final output signals, y1 and y2 */
    if ((y1 = calloc((SIGNAL_LENGTH + BLOCK_LENGTH), sizeof(complex))) == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(8);
    }
    if ((y2 = calloc((SIGNAL_LENGTH + BLOCK_LENGTH), sizeof(complex))) == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(8);
    }
    fprintf(stderr, "Rebuilding Output Vector\n");
    /* ------------------------------------------------------------------ */
    /* REBUILD OUTPUT VECTORS                                              */
    /* ------------------------------------------------------------------*/
    /* Inverse Fourier Transform and synthesize the separated signals */
    block = BETA * BLOCK_LENGTH;
    for (i = 0; i < NUMBER_BLOCKS; i++) {
for(j = 0; j < BLOCK_LENGTH; j++) {
    /*Retrieve seperated signals from final stage */
    yy1[j] = x1[j][yy_counter];
    /*Retrieve seperated signals from final stage */
    yy2[j] = x2[j][yy_counter];
}
yy_counter++;
        ifft(BLOCK_LENGTH, yy1);  /* Inverse Fourier Transform */
        ifft(BLOCK_LENGTH, yy2);  /* Inverse Fourier Transform */
        for(j = 0; j < BLOCK_LENGTH; j++) {
    /* Pull out the real part of yy1, but keep in complex
     * form with 0 for imaginary part                        */
    yy1[j] = cmplx(real(yy1[j]),0);
    /* Pull out the real part of yy2, but keep in complex
     * form with 0 for imaginary part                           */
    yy2[j] = cmplx(real(yy2[j]),0);
        }
        for(j = (L1 - block ); j < L1; j++) {
    /* First part of reconstructed block */
            y1[j] = cadd(y1[j],yy1[y_counter]);
    /* First part of reconstructed block */
    y2[j] = cadd(y2[j],yy2[y_counter]);
            y_counter++;
        }
        for(j = L1; j < (L1 + HALF_BLOCK_LENGTH); j++) {
            y1[j] = yy1[y_counter];  /* Second part of reconstructed block */
            y2[j] = yy2[y_counter];  /* Second part of reconstructed block */
            y_counter++;
        }
        L1 = L1 + HALF_BLOCK_LENGTH;
        y_counter = 0;
    }

    fprintf(stderr, "Normalizing\n");
    /* ------------------------------------------------------------------------*/
    /* NORMALIZATION                                                           */
    /* ------------------------------------------------------------------------*/
    /* Normalization code, this will need to be modified for real-time
     *implementation                                                     */
    /* Find the largest sample out of each signal */
    for (i = 0; i < (SIGNAL_LENGTH + BLOCK_LENGTH); i++) {
if(fabs(real(y1[i])) > max_sample_value_1) {
    max_sample_value_1 = fabs(real(y1[i]));
}
if(fabs(real(y2[i])) > max_sample_value_2) {
    max_sample_value_2 = fabs(real(y2[i]));
}
    }
    /* Do one divide to avoid many divides, and add in an offset */
    max_sample_value_1 = 1 / (max_sample_value_1 + E);
    max_sample_value_2 = 1 / (max_sample_value_2 + E);
    /* Normalize */
    for (i = 0; i < (SIGNAL_LENGTH + BLOCK_LENGTH); i++) {
y1[i].x = y1[i].x * max_sample_value_1;
y2[i].x = y2[i].x * max_sample_value_2;
    }
    /* ------------------------------------------------------------------------*/
    /* Open our output files */
    if ((output_file1 = fopen("output_source1.txt", "w")) == NULL) {
fprintf(stderr, "Output file 1 cannot be opened");
exit(8);
    }
    if ((output_file2 = fopen("output_source2.txt", "w")) == NULL) {
fprintf(stderr, "Output file 2 cannot be opened");
exit(8);
    }
    /* Save separated signals */
    for(i = 0; i < OUTPUT_SIGNAL_LENGTH; i++) {
fprintf(output_file1, "%0.16f\n", y1[i].x);
fprintf(output_file2, "%0.16f\n", y2[i].x);
    }

    /* Close our output files */
    fclose(output_file1);
    fclose(output_file2);

    /* Final cleanup before we exit */
    for(i = 0; i < NUMBER_BLOCKS; i++) {
free(x1[i]);
free(x2[i]);
    }
    free(y1); free(y2);
    free(yy1); free(yy2);
    free(Y1); free(Y2);

    return(0);
}
```

complex.c

```
/* complex.c - complex arithmetic functions */

#include <math.h>                         /* for MSC and TC/BC, it declares: */
                                          /* \ttt{struct complex} and
                                           * \ttt{cabs()} */
struct complex {double x, y;};            /* uncomment if not MSC or TC/BC */

                                          /* uncomment if not MS or TC/BC */
//  double cabs(z)
//  complex z;
//  {
//      return sqrt(z.x * z.x + z.y * z.y);
//  }


typedef struct complex complex;

complex cmplx(x, y)                             /* z = cmplx(x,y) = x+jy */
double x, y;
{
        complex z;

        z.x = x;  z.y = y;

        return z;
}

complex conjg(z)                                /* complex conjugate of z=x+jy*/
complex z;
{
        return cmplx(z.x, -z.y);                /* returns z* = x-jy */
}

complex cadd(a, b)                              /* complex addition */
complex a, b;
{
        return cmplx(a.x + b.x, a.y + b.y);
}

complex csub(a, b)                              /* complex subtraction */
complex a, b;
{
        return cmplx(a.x - b.x, a.y - b.y);
}

complex cmul(a, b)                              /* complex multiplication */
complex a, b;
{
        return cmplx(a.x * b.x - a.y * b.y, a.x * b.y + a.y * b.x);
}

complex rmul(a, z)                              /* multiplication by real */
double a;
complex z;
{
        return cmplx(a * z.x, a * z.y);
```

```
}

complex cdiv(a, b)                                 /* complex division */
complex a, b;
{
   double D = b.x * b.x + b.y * b.y;

   return cmplx((a.x * b.x + a.y * b.y) / D, (a.y * b.x - a.x * b.y) / D);
}

complex rdiv(z, a)                                 /* division by real */
complex z;
double a;
{
       return cmplx(z.x / a, z.y / a);
}

double real(z)                                     /* real part Re(z) */
complex z;
{
       return z.x;
}

double aimag(z)                                    /* imaginary part Im(z) */
complex z;
{
       return z.y;
}

complex cexp(z)                                    /* complex exponential */
complex z;
{
       double R = exp(z.x);

       return cmplx(R * cos(z.y), R * sin(z.y));
}
```

fft.c

```
/* fft.c --  in-place decimation-in-time FFT */

#include "cmplx.h"

void shuffle(), dftmerge();
void fft(N, X)
complex *X;
int N;
{
       shuffle(N, X);
       dftmerge(N, X);
}
```

shuffle.c

```
/* shuffle.c - in-place shuffling (bit-reversal) of a complex array */

#include "cmplx.h"

void swap();
int bitrev();

void shuffle(N, X)
complex *X;
int N;                                     /* \(N\) must be a power of 2 */
{
       int n, r, B=1;

       while ( (N >> B) > 0 )              /* \(B\) = number of bits */
              B++;

       B--;                                /* \(N = 2\sp{B}\) */

       for (n = 0; n < N; n++) {
           r = bitrev(n, B);               /* bit-reversed version of \(n\) */
           if (r < n) continue;            /* swap only half of the \(n\)s */
           swap(X+n, X+r);                 /* swap by addresses */
           }
}
```

dftmerge.c

```
/* dftmerge.c - DFT merging for radix 2 decimation-in-time FFT */

#include "cmplx.h"

void dftmerge(N, XF)
complex  *XF;
int N;
```

```
{
        double pi = 4. * atan(1.0);
        int k, i, p, q, M;
        complex  A, B, V, W;

        M = 2;
        while (M <= N) {                                 /* two \((M/2)\)-DFTs into one
                                                          * \(M\)-DFT  */
              W = cexp(cmplx(0.0, -2 * pi / M));         /* order-\(M\) twiddle factor */
              V = cmplx(1., 0.);                         /* successive powers of \(W\) */
              for (k = 0; k < M/2; k++) {                /* index for an \((M/2)\)-DFT */
                    for (i = 0; i < N; i += M) {         /* \(i\)th butterfly; increment
                                                          * by \(M\) */
                          p = k + i;                     /* absolute indices for */
                          q = p + M / 2;                 /* \(i\)th butterfly */
                          A = XF[p];
                          B = cmul(XF[q], V);            /* \(V = W\sp{k}\) */
                          XF[p] = cadd(A, B);            /* butterfly operations */
                          XF[q] = csub(A, B);
                          }
                    V = cmul(V, W);                      /* \(V = VW = W\sp{k+1}\) */
                    }
              M = 2 * M;                                 /* next stage */
              }
}
```

swap.c

```
/*  swap.c  --  swap two complex numbers (by their addresses)  */

#include "cmplx.h"

void swap(a,b)
complex *a, *b;
{
        complex t;

         t = *a;
        *a = *b;
        *b =  t;
}
```

bitrev.c

```
/* bitrev.c - bit reverse of a B-bit integer n */
```

```
#define two(x)       (1 << (x))                   /*\(2\sp{x}\) by left-shifting*/

int bitrev(n, B)
int n, B;
{
       int m, r;

       for (r=0, m=B-1; m>=0; m--)
          if ((n >> m) == 1) {                    /* if \(2\sp{m}\) term is
                                                   * present, then */
              r += two(B-1-m);                    /* add \(2\sp{B-1-m}\) to
                                                   * \(r\), and */
              n -= two(m);                        /* subtract \(2\sp{m}\)
                                                   * from \(n\) */
              }

       return(r);
}
```

ifft.c

```
/* ifft.c - inverse FFT */

#include "cmplx.h"

void fft();

void ifft(N, X)
complex *X;
int N;
{
    int k;

    for (k=0; k<N; k++)
         X[k] = conjg(X[k]);                      /* conjugate input */

    fft(N, X);                                    /* compute FFT of conjugate */

    for (k=0; k<N; k++)
         X[k] = rdiv(conjg(X[k]), (double)N);     /* conjugate and divide by
                                                   * \(N\) */
}
```

cmplx.h

```
/* cmplx.h - complex arithmetic declarations */

#include <math.h>                         /* in MSC and TC/BC, it declarares: */
                                          /* \ttt{struct complex} and
                                           * \ttt{cabs(z)} */

struct complex{double x, y;};             /* uncomment if neccessary */
double cabs(struct complex);              /* uncomment if neccesary */

typedef struct complex complex;

complex cmplx(double, double);            /* define complex number */
complex conjg(complex);                   /* complex conjugate */

complex cadd(complex, complex);           /* complex addition */
complex csub(complex, complex);           /* complex subtraction */
complex cmul(complex, complex);           /* complex multiplication */
complex cdiv(complex, complex);           /* complex division */

complex rmul(double, complex);            /* multiplication by real */
complex rdiv(complex, double);            /* division by real */

double real(complex);                     /* real part */
double aimag(complex);                    /* imaginary part */

complex cexp(complex);                    /* complex exponential */
```

List of Symbols, Abbreviations, and Acronyms

$\mathbf{A}^H$	–	Complex-conjugate transpose
$\mathbf{A}^T$	–	Transpose
$\mathbf{A}^*$	–	Complex conjugate
$\mathbf{C}$	–	Correction matrix
$\mathbf{F}$	–	Discrete Fourier transform matrix
$\mathbf{G}$	–	Projection matrix
$\mathbf{H}$	–	Mixing filter matrix
$\mathbf{h}_{ji}$	–	Room impulse response from source i to microphone j
$E[\cdot]$	–	Expectation operator
k	–	Superblock index
K	–	Number of superblocks
L	–	Number of adaptive iterations
m	–	Block index
M	–	Number of blocks in each superblock
n	–	Discrete time index
N	–	Length of input signal in sample
P	–	Mixing filter length
$\mathbf{P}$	–	Permutation matrix
q	–	Stage index
Q	–	Un-mixing filter length
$\mathbf{R_x}$	–	Correlation, covariance matrix of $\mathbf{x}$
$\mathbf{s}$	–	Vector of sources
SG	–	Number of multistages
T	–	Block length
^{T}	–	Matrix transpose operator
U	–	Number of blocks
$\mathbf{V}$	–	Cross-power spectral density matrix
$\mathbf{W}$	–	Un-mixing filter matrix
x_i	–	i^{th} component of vector, $\mathbf{x}$
$\mathbf{x}$	–	Vector of mixtures
$\mathbf{y}$	–	Vector of separated signals
$\mathbf{Z}$	–	Diagonal matrix with $Z_{ii} = 1$ for $i < Q$ and $Z_{ii} = 0$ for $i \geq Q$

$\text{Off}(\cdot)$	–	Off Diagonal matrix
$*$	–	convolution operator
$\hat{}$	–	estimate
$\|\|\cdot\|\|_F^2$	–	Squared Frobenius norm

β	–	Window overlap factor
$\tilde{\mu}$	–	Normalized step size
$\tilde{\rho}$	–	Reflectivity coefficient
ω	–	Discrete frequency index

ASR	–	Automatic Speech Recognition
BSS	–	Blind Source (or Speech) Separation
CPSD	–	CrossPower Spectral Density
dB	–	Decibel
DIT	–	Decimation in Time
DFT	–	Discrete Fourier Transform
DSP	–	Digital Signal Processing (or Processor)
FDADF	–	FrequencyDomain Adaptive Decorrelation Filtering
FDSOS	–	FrequencyDomain, Second Order Statistics
FFT	–	Fast Fourier Transform
LDC	–	Linguistics Data Consortium
LSI	–	Linear, Shift-Invariant
MRFD	–	Multiresolution Frequency Domain
PSD	–	Power Spectral Density
SIR	–	Signal-to-Interference Ratio
SIRI	–	Signal-to-Interference Ratio Improvement
$\mathrm{SIR_i}$	–	Input Signal-to-Interference Ratio
$\mathrm{SIR_o}$	–	Output Signal-to-Interference Ratio
SNR	–	SignaltoNoise Ratio
SNRI	–	SignaltoNoise Ratio Improvement
STFT	–	Short-Time Fourier Transform
TIMIT	–	Texas Instruments, Massachusetts Institute of Technology
WGN	–	White Gaussian Noise